

Nucleus C++ PLUS

Reference Manual



0001127-001 Rev 100

Copyright (c) 2002
Accelerated Technology, Inc.
720 Oak Circle Dr. E.
Mobile, AL 36609
(251) 661-5770



Related Documentation

Nucleus PLUS Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus PLUS kernel.

Nucleus PLUS Internals, by Accelerated Technology, describes, in considerable detail, the implementation of the Nucleus PLUS kernel.

Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - `Courier New`

Filenames - `COURIER NEW, ALL CAPS`

Interactive Command Lines - **`Courier New, Bold`**

Menu Items/Buttons – *Times New Roman Italic*

Trademarks

MS-DOS is a trademark of Microsoft Corporation

UNIX is a trademark of X/Open

IBM PC is a trademark of International Business Machines, Inc.

Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology

720 Oak Circle Drive, East

Mobile, AL 36609

800-468-6853

251-661-5770

251-661-5788 (fax)

support@atinucleus.com

<http://www.acceleratedtechnology.com>

Copyright (©) 2002, All Rights Reserved.

Document Part Number : 0001127-001 Rev 101

Last Revised: March 8, 2001





Contents

Chapter 1 -Nucleus C++ PLUS User's	15
About this Manual	16
What is in this User's Guide?	16
What is Nucleus C++ PLUS?	16
Tasks	16
Inter-task Communication	16
Periodic Timers.....	17
Events	17
Semaphores	17
External Interrupts	17
Software Interrupts	17
Memory Management	17
Nucleus C++ PLUS Class Hierarchy	18
Nucleus C++ PLUS Source Files.....	19
Portable Source Files	20
Target Dependent Source Files	21
Application Dependent Source Files.....	21
Optimizations for Embedded Systems	21
Task Terminator.....	22
'Helper' Threads	22
IODriver.....	22
Signal	22
NewAware	23
The Minimal Configuration	23
Task.....	24
ClassTask Overview	24
Class Structure	27
Creating Your Own Task, 1-2-3	28
Operating on Tasks	31
Accessing Task Information	34



Inter-task Communication.....	36
Class Q Overview	36
Structure of Class Q	37
Understanding Class Q Behavior	38
Using Queues, 1-2-3	39
Declare the Message Type	39
Declare and Define the Typed Apple Queue Class	40
Declaring Tasks that Communicate to Each Other	41
Creating the Application Task Instances	43
Define the Behavior of the Quarterback Task	44
Define the Behavior of the Receiver Task	45
What is the Difference Between the Queues?	47
Copy-Oriented Queues	47
Heterogeneous Queues and Polymorphism	47
Using Class Q Directly	48
Accessing Queue Information	48
Periodic Timers	50
Class Timer Overview	50
Structure of Class Timer	51
Using Timers, 1-2-3	51
Accessing Timer Information	59
Events.....	60
Class EventGroup Overview	60
Structure of Class EventGroup	61
Class Event Overview	61
Structure of Class Event	61
Using Events, 1-2-3	62
Continuing from the Previous Example	62
Modify the Quarterback Classes	62
Modify the Quarterback Task Behavior	63
Modify the Quarterback Timer Behavior	64
Events and Suspension	66
Using Event Groups, 1-2-3	66
Declare Some Bit Meanings	66
Do Something and Set the Event Flags	67
Check the Status of the Event Flags	68
Accessing EventGroup Information	68
Semaphores	70
Class Semaphore Overview	70
Structure of Class Semaphore	72
Structure of Class PrioritySemaphore	72
Structure of Class FifoSemaphore	72



What is a Semaphore?	73
The Problem	73
The Solution	73
Deciding the Order of Suspension	74
Using Semaphores, 1-2-3	74
Add a Semaphore Object to the Class	74
Use the Semaphore to Protect Shared Data	75
Using Suspension Parameters	76
Using the Counting Features	76
Accessing Semaphore Information	77
External Interrupts	78
Class LowLevelInterrupt Overview	78
Structure of Class LowLevelInterrupt	78
Class HighLevelInterrupt Overview	79
Structure of Class HighLevelInterrupt	79
Using Interrupts, 1-2-3	79
Example Application Overview	80
Example Application Class Structures	80
Example Application Sequence	82
Interrupt Simulation	84
Interrupt Application Source Code Details	86
Creating the Objects	94
Accessing HighLevelInterrupt Information	95
Software Interrupts	96
Class Signal Overview	96
Structure of Class Signal	97
Using Software Interrupts, 1-2-3	97
Subclass class Signal	98
Define Signal Handler Behavior	99
Create a Simple Quarterback and Receiver of Signals	99
Create Objects	104
Special Memory Management Classes	107
Class MemoryPool Overview	107
Structure of Class MemoryPool	108
Using Memory Pools, 1-2-3	108
Accessing MemoryPool Information	110
Class PartitionPool Overview	111
Structure of Class PartitionPool	113
Using Partition Pools, 1-2-3	113
Accessing PartitionPool Information	114



Chapter 2 - Class Reference	117
class Box : public NucleusPlus	118
Box::~~Box	120
Box::Box	121
Box::Broadcast	122
Box::Receive	123
Box::Receive	124
Box::Reset	125
Box::Reset	126
Box::Send	127
Box::UpdateInfo	129
class BoxInfo : public SuspendInfo	130
BoxInfo::~~BoxInfo	131
BoxInfo::BoxInfo	132
BoxInfo::GetControlBlock	133
BoxInfo::IsMessage	134
BoxInfo::Update	135
class CommunicationInfo : public SuspendInfo	136
CommunicationInfo::~~CommunicationInfo	137
CommunicationInfo::CommunicationInfo	138
CommunicationInfo::GetAvailable	139
CommunicationInfo::GetBufferAddress	140
CommunicationInfo::GetCurrent	141
CommunicationInfo::GetMessageSize	142
CommunicationInfo::GetMessageType	143
CommunicationInfo::GetSize	144
class DevelopmentService	145
DevelopmentService::~~DevelopmentService	146
DevelopmentService::DevelopmentService	147
DevelopmentService::DisableHistorySaving	148
DevelopmentService::EnableHistorySaving	149
DevelopmentService::LicenseInformation	149
DevelopmentService::LicenseInformation	150
DevelopmentService::MakeHistoryEntry	151
DevelopmentService::ReleaseInformation	153
DevelopmentService::RetrieveHistoryEntry	154
class Event	156
Event::~~Event	157
Event::Clear	158
Event::Event	159
Event::Initialize	160
Event::Retrieve	161
Event::Set	163



class EventGroup : public NucleusPlus	164
class EventGroupInfo : public SuspendInfo	164
EventGroupInfo::~EventGroupInfo	165
EventGroupInfo::EventGroupInfo	166
EventGroupInfo::GetFlags	168
EventGroupInfo::Update	169
class HighLevelInterrupt : public NucleusPlus	170
HighLevelInterrupt::~HighLevelInterrupt	171
HighLevelInterrupt::Activate	172
HighLevelInterrupt::Current	173
HighLevelInterrupt::Entry	174
HighLevelInterrupt::GetHighLevelInterrupt	175
HighLevelInterrupt::HighLevelInterrupt	176
HighLevelInterrupt::UpdateInfo	178
class HighLevelInterruptInfo : public NucleusPlusInfo	179
HighLevelInterruptInfo::~HighLevelInterruptInfo	180
HighLevelInterruptInfo::GetControlBlock	181
HighLevelInterruptInfo::HighLevelInterruptInfo	182
HighLevelInterruptInfo::Update	183
class InterruptSystem	184
InterruptSystem::~InterruptSystem	185
InterruptSystem::Disable	186
InterruptSystem::Enable	187
InterruptSystem::InterruptSystem	188
InterruptSystem::SetupVector	189
class LowLevelInterrupt	191
LowLevelInterrupt::~LowLevelInterrupt	192
LowLevelInterrupt::Entry	193
LowLevelInterrupt::GetVector	194
LowLevelInterrupt::Initialize	195
LowLevelInterrupt::LowLevelInterrupt	196
class MemoryPool : public NucleusPlus	198
MemoryPool::~MemoryPool	199
MemoryPool::Allocate	200
MemoryPool::Deallocate	202
MemoryPool::MemoryPool	203
MemoryPool::UpdateInfo	205
class MemoryPoolInfo : public PoolInfo	206
MemoryPoolInfo::~MemoryPoolInfo	207
MemoryPoolInfo::GetControlBlock	208
MemoryPoolInfo::GetMinimumAllocation	209
MemoryPoolInfo::MemoryPoolInfo	210
MemoryPoolInfo::Update	211



class NppPLUS : public NppComponent	212
NppPLUS::~~NppPLUS	213
NppPLUS::Initialize	214
NppPLUS::InitializeBox	215
NppPLUS::InitializeEvent	216
NppPLUS::InitializeEventGroup	217
NppPLUS::InitializeHelpers	218
NppPLUS::InitializeHighLevelInterrupt	219
NppPLUS::InitializeIODriver	220
NppPLUS::InitializeLowLevelInterrupt	221
NppPLUS::InitializeMemoryPool	222
NppPLUS::InitializePartitionPool	223
NppPLUS::InitializePipe	224
NppPLUS::InitializeQ	225
NppPLUS::InitializeSemaphore	226
NppPLUS::InitializeTask	227
NppPLUS::InitializeTimer	228
NppPLUS::NppPLUS	229
class NppProtect : public NucleusPlus	230
Public Member Functions	230
NppProtect::~~NppProtect	231
NppProtect::GetProtectedBYTE_PTR	232
NppProtect::GetProtectedCHAR	233
NppProtect::GetProtectedINT	234
NppProtect::GetProtectedUNSIGNED	235
NppProtect::GetProtectedUNSIGNED_PTR	237
NppProtect::NppProtect	239
NppProtect::Protect	240
NppProtect::SetProtectedBYTE_PTR	241
NppProtect::SetProtectedCHAR	242
NppProtect::SetProtectedINT	243
NppProtect::SetProtectedUNSIGNED	244
NppProtect::SetProtectedUNSIGNED_PTR	245
NppProtect::UnProtect	246
class PartitionPool : public NucleusPlus	247
PartitionPool::~~PartitionPool	249
PartitionPool::Allocate	250
PartitionPool::Deallocate	252
PartitionPool::PartitionPool	254
PartitionPool::UpdateInfo	255
class PartitionPoolInfo : public PoolInfo	256
PartitionPoolInfo::~~PartitionPoolInfo	257
PartitionPoolInfo::GetControlBlock	258
PartitionPoolInfo::GetPartitionsAllocated	259
PartitionPoolInfo::GetPartitionSize	260
PartitionPoolInfo::PartitionPoolInfo	261
PartitionPoolInfo::Update	262



class Pipe : public NucleusPlus.....	263
Pipe::~Pipe.....	265
Pipe::Broadcast.....	266
Pipe::Pipe.....	268
Pipe::Receive.....	270
Pipe::Reset.....	272
Pipe::Send.....	274
Pipe::SendToFront.....	276
Pipe::UpdateInfo.....	278
class PipeInfo : public CommunicationInfo.....	279
PipeInfo::~PipeInfo.....	280
PipeInfo::GetControlBlock.....	281
PipeInfo::PipeInfo.....	282
PipeInfo::Update.....	283
class PoolInfo : public SuspendInfo.....	284
PoolInfo::~PoolInfo.....	285
PoolInfo::GetBytesAvailable.....	286
PoolInfo::GetPoolAddress.....	287
PoolInfo::GetPoolSize.....	288
PoolInfo::PoolInfo.....	289
class Q : public NucleusPlus.....	290
Q::~Q.....	291
Q::Broadcast.....	292
Q::Q.....	294
Q::Receive.....	296
Q::Reset.....	299
Q::Send.....	300
Q::SendToFront.....	302
Q::UpdateInfo.....	304
class QInfo : public CommunicationInfo.....	305
QInfo::~QInfo.....	306
QInfo::GetControlBlock.....	307
QInfo::QInfo.....	308
QInfo::Update.....	309
class Semaphore : public NucleusPlus.....	310
Semaphore::~Semaphore.....	311
Semaphore::Obtain.....	312
Semaphore::Release.....	314
Semaphore::Reset.....	315
Semaphore::Semaphore.....	316
Semaphore::UpdateInfo.....	317
class SemaphoreInfo : public SuspendInfo.....	318
SemaphoreInfo::~SemaphoreInfo.....	319
SemaphoreInfo::GetControlBlock.....	320
SemaphoreInfo::GetCount.....	321
SemaphoreInfo::SemaphoreInfo.....	322
SemaphoreInfo::Update.....	323



class Signal.....	324
Signal::~Signal.....	325
Signal::Disable.....	326
Signal::Enable.....	327
Signal::Send.....	328
Signal::Signal.....	329
Signal::Signal.....	333
Signal::SignalHandler.....	334
class SuspendInfo : public NucleusPlusInfo.....	335
SuspendInfo::~SuspendInfo.....	336
SuspendInfo::GetFirstTaskWaiting.....	337
SuspendInfo::GetNumberTasksWaiting.....	338
SuspendInfo::GetSuspendType.....	339
SuspendInfo::SuspendInfo.....	340
class SystemClock.....	341
SystemClock::~SystemClock.....	342
SystemClock::Get.....	343
SystemClock::Retrieve.....	344
SystemClock::Set.....	345
SystemClock::SystemClock.....	346
class Task : public NucleusPlus.....	347
Task::~Task.....	348
Task::ChangePriority.....	349
Task::ChangeTimeSlice.....	350
Task::CheckStack.....	351
Task::Current.....	352
Task::Entry.....	353
Task::GetTask.....	354
Task::IsCurrent.....	355
Task::PreemptionOff.....	356
Task::PreemptionOn.....	357
Task::Relinquish.....	358
Task::Reset.....	359
Task::Resume.....	362
Task::Sleep.....	363
Task::Start.....	365
Task::Suspend.....	366
Task::Task.....	367
Task::Terminate.....	369
Task::UpdateInfo.....	371
class TaskInfo : public ThreadInfo.....	373
TaskInfo::~TaskInfo.....	374
TaskInfo::GetControlBlock.....	375
TaskInfo::GetPreemption.....	376
TaskInfo::GetTaskStatus.....	377
TaskInfo::GetTimeslice.....	378



TaskInfo::TaskInfo	379
TaskInfo::Update	380
class ThreadInfo : public NucleusPlusInfo	381
ThreadInfo::~ThreadInfo	382
ThreadInfo::GetMinimumStackSize	383
ThreadInfo::GetPriority	384
ThreadInfo::GetStackAddress	385
ThreadInfo::GetStackSize	386
ThreadInfo::GetTimesScheduled	387
ThreadInfo::ThreadInfo	388
class Timer : public NucleusPlus	389
Timer::~Timer	390
Timer::Disable	391
Timer::Enable	392
Timer::ExpirationRoutine	393
Timer::Reset	394
Timer::Timer	396
Timer::UpdateInfo	397
class TimerInfo : public NucleusPlusInfo	398
TimerInfo::~TimerInfo	399
TimerInfo::GetControlBlock	400
TimerInfo::GetExpirations	401
TimerInfo::GetId	402
TimerInfo::GetInitialTime	403
TimerInfo::GetRescheduleTime	404
TimerInfo::IsEnabled	405
TimerInfo::TimerInfo	406
TimerInfo::Update	407







Nucleus C++ PLUS User's Guide

About this Manual
What is C++ PLUS?
Nucleus C++ PLUS Class Hierarchy
Nucleus C++ PLUS Source Files
Optimization for Embedded Systems
Tasks
InterTask Communication
Periodic Timers
Events
Semaphores
External Interrupts
Software Interrupts
Memory Management



About this Manual

This manual is intended to give readers a high level overview of Nucleus C++ PLUS so they are better prepared to take advantage its of specific design goals and features.

What is in this User's Guide?

This guide contains explanations of the various source files, what demos are included, application startup, and the various classes included in Nucleus C++ PLUS.

What is Nucleus C++ PLUS?

Since the C++ programming language does not support the concept of multitasking, special C++ support software is required for use in an environment like Nucleus (*embedded, real-time, pre-emptive multi-tasking*).

The Nucleus C++ PLUS package includes components that contain object-oriented C++ class interfaces into the services provided by the Nucleus PLUS real-time kernel. These interfaces help programmers deal with inherent complexities associated with developing real-time embedded multitasking systems.

Nucleus C++ PLUS models traditional RTOS services as C++ classes. This allows the various real-time elements in an embedded system to be easily managed using object-oriented techniques. You can easily create reusable embedded design patterns that include *threads, interrupts, messaging, events, signals, timers, semaphores, and mutexes*.

The advantage to Nucleus C++ is *portability* across a wide selection of Nucleus C++ embedded processor targets. Since reuse is maximized, it is easier to leverage prior work in future applications.

Tasks

The Nucleus C++ PLUS class `Task` is a base class used to create and manage multiple threads in an embedded application. The class is implemented using Nucleus PLUS real-time task services. Programmers either create new thread behaviors or reuse existing behaviors that solve related tasks.

Inter-task Communication

The Nucleus C++ PLUS classes `Pipe`, `Q`, and `Box` are used for inter-task communication. For real-time behavior, they have been implemented using Nucleus PLUS inter-task communication services.



Periodic Timers

Using Nucleus PLUS timers, the Nucleus C++ PLUS class `Timer` provides a C++ class interface for supporting real-time periodic timers. Programmers either create new timer expiration behaviors or reuse existing behaviors that solve related duties.

Events

The Nucleus C++ PLUS event classes are used to create real-time architectures that are event-driven, providing a commonly required synchronization mechanism. The Nucleus C++ classes are `Event` and `EventGroup`. Event groups support the ability to suspend waiting for a logical combination of multiple events to occur.

Semaphores

The Nucleus C++ PLUS classes `Semaphore`, `PrioritySemaphore`, and `FifoSemaphore` are used to manage multithreaded access to critical shared resources. These critical section management classes are implemented using Nucleus PLUS semaphore services.

External Interrupts

The Nucleus C++ PLUS classes `LowLevelInterrupt` and `HighLevelInterrupt` are base classes that manage asynchronous external interrupts in an embedded application. The classes demonstrate the power of *portability*. The various interrupt architectures found in today's advanced embedded processors are very different but the Nucleus C++ PLUS classes do not across all supported Nucleus C++ embedded processor targets.

Software Interrupts

The Nucleus C++ PLUS class `Signal` is a class that manages asynchronous software interrupts in an embedded application using the Nucleus PLUS signaling services. This is a synchronization mechanism used for inter-task communication. The class `Task` has special software to support this advanced mechanism when enabled.

Memory Management

The Nucleus C++ BASE package includes a re-entrant, real-time solution for the standard C++ memory operators `new` and `delete`. Nucleus C++ PLUS extends this and adds classes `MemoryPool` and `PartitionPool`. These classes encapsulate Nucleus PLUS memory management services.

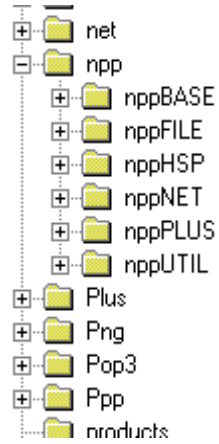


[illegible]

Nucleus C++ PLUS Class Hierarchy

Nucleus C++ PLUS Source Files

Nucleus C++ PLUS source files are delivered in the \NPPPLUS sub-directory under the Nucleus C++ npp directory.



Nucleus C++ PLUS directory is nppPLUS



Portable Source Files

In the \NPPPLUS sub-directory you will find the following portable files. All Nucleus C++ PLUS targets share these files and they are maintained on a product wide basis.

File	Description
NPPBOX.H NPPBOX.CPP NPPBOX.INL	Source files for classes: Box.
NPPDEVL.H NPPDEVL.CPP NPPDEVL.INL	Source files for classes: DevelopmentService.
NPPEVNT.H NPPEVNT.CPP NPPEVNT.INL	Source files for classes: EventGroup; Event.
NPPIODR.H NPPIODR.CPP NPPIODR.INL	Source files for classes: IODriver.
NPPISR.H NPPISR.CPP NPPISR.INL	Source files for classes: LowLevelInterrupt; HighLevelInterrupt.
NPPPIPE.H NPPPIPE.CPP NPPPIPE.INL	Source files for classes: Pipe.
NPPPLUS.H NPPPLUS.CPP NPPPLUS.INL	Source files for classes: NppPLUS.
NPPPMEM.H NPPPMEM.CPP NPPPMEM.INL	Source files for classes: MemoryPool; PartitionPool.
NPPPROT.H NPPPROT.CPP NPPPROT.INL	Source files for classes: NppProtect.
NPPQ.H NPPQ.CPP NPPQ.INL	Source files for classes: Q.
NPPSYNC.H NPPSYNC.CPP NPPSYNC.INL	Source files for classes: Semaphore; FifoSemaphore; PrioritySemaphore.
NPPTASK.H NPPTASK.CPP NPPTASK.INL	Source files for classes: Task; Signal.
NPPTIMR.H NPPTIMR.CPP NPPTIMR.INL	Source files for classes: Timer; SystemClock.



Target Dependent Source Files

There are no target dependent files in Nucleus C++ PLUS. If for some reason a particular version of Nucleus C++ requires special software, the convention is that the file will be named `NPPPLUSS.H`, `NPPPLUSS.INL`, and `NPPPLUSS.CPP` and it will be located in the `nppPLUS` sub-directory.

Application Dependent Source Files

In the `\NPPBASE` directory you will find an application dependent file, `NPPAPP.H`. Each Nucleus C++ application will have a different version of this file. It is used to specify application specific parameters for all Nucleus C++ components used in that particular application.

File	Description
<code>NPPAPP.H</code>	This file contains all application tuning settings for Nucleus C++. By setting various preprocessor switches, you can include or exclude various features from the build to minimize the footprint of your application.

Optimizations for Embedded Systems

There are options within the application tuning header file, `NPPAPP.H`, that allow you to further customize your Nucleus C++ PLUS application by cutting down on what features are included in the build.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_TASK_TERMINATION_COORDINATOR</code>	Set to 1 to provide the task termination semaphore data member for class <code>Task</code> .
<code>NU_HELPERS</code>	Set to 1 to support task self-delete, self-terminate, and self-reset.
<code>NU_TASK_IODRIVER_SUPPORT</code>	Set to 1 to include support for class <code>IODriver</code> in class <code>Task</code> .
<code>NU_TASK_SIGNAL_SUPPORT</code>	Set to 1 if using <code>Signal</code> .
<code>NU_NEW_AWARE</code>	Set to 1 if you want tasks to be aware of whether or not they have been dynamically allocated via the new operator.



Task Terminator

The Nucleus C++ PLUS class `Task` contains a data member that is a semaphore used for task termination coordination. This support requires resources for every derived class `Task` object in the system. To reduce base application resource requirements, this support data member is now optional.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_TASK_TERMINATION_COORDINATOR</code>	Set to 1 to provide the task termination semaphore data member for class <code>Task</code> .

'Helper' Threads

Nucleus C++ PLUS contains support for the concept of task self-delete, self-terminate, and self-reset. This requires the presence of helper threads that require system resources. This support is optional to reduce system resource usage for applications that do not require this advanced feature.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_HELPERS</code>	Set to 1 to support <code>Task</code> self-delete, self-terminate, and self-reset.

IODriver

Class `IODriver` has members to coordinate suspending and resuming a task using it. If `IODriver` is not used, support can be removed.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_TASK_IODRIVER_SUPPORT</code>	Set to 1 to include support for class <code>IODriver</code> in class <code>Task</code> .

Signal

Class `Task` has built in support for up to 30 registered signal handlers. If class `Signal` is not used, support can be removed by defining the parameter to 0.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_TASK_SIGNAL_SUPPORT</code>	Set to 1 to include support for class <code>Signal</code> in class <code>Task</code> .



NewAware

Class `NewAware` is a base class that gives any derived class the ability to know if it was allocated dynamically via the `new` operator. If it is included in the build, class `Task` is multiply inherited from classes `NucleusPlus` and `NewAware`. This gives it the ability to know if it was *new*'ed or not. Defining `NU_NEW_AWARE` to 0 removes `NewAware` and the multiple-inheritance.

NOTE: EC++ compliance requires this setting to be 1.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_NEW_AWARE</code>	Set to 1 if you want <code>Tasks</code> to be aware of whether or not they have been dynamically allocated via the <code>new</code> operator.

The Minimal Configuration

You can make your application the minimal configuration in terms of memory usage by simply setting all of these switches in `NPPAPP.H` to 0.

Parameter in <code>NPPAPP.H</code>	Meaning
<code>NU_TASK_TERMINATION_COORDINATOR</code>	0
<code>NU_HELPERS</code>	0
<code>NU_TASK_IODRIVER_SUPPORT</code>	0
<code>NU_TASK_SIGNAL_SUPPORT</code>	0
<code>NU_NEW_AWARE</code>	0



Task

The Nucleus C++ PLUS class `Task` is a base class used to create and manage multiple threads in an embedded application. The class is implemented using Nucleus PLUS real-time task services. Programmers either create new thread behaviors or reuse existing behaviors that solve related tasks.

ClassTask Overview

A task is a semi-independent program segment with a dedicated purpose. Most modern real-time applications require multiple tasks. Additionally, these tasks often have varying degrees of importance. Managing the execution of competing, real-time tasks is the main purpose of Nucleus C++.

Task States

Each task is always in one of five states: executing, ready, suspended, terminated, or finished. The following list describes each of the task states.

Task Name	Description
Executing	Task is currently running.
Ready	Task is ready, but another task is currently running.
Suspended	Task is dormant while waiting for the completion of a service request. When the request is complete, the task is moved to the ready state.
Terminated	Task was killed. Once in this state, the task cannot execute again until it is reset.
Finished	Task finished its processing and returned from initial entry routine. Once in this state, the task cannot execute again until it is reset.



Priority

The importance of a Nucleus C++ task is determined by its user-assigned priority. Task priorities are numerical values that range from 0 to 255. Lower numerical values indicate higher priority. For example, a task with a numerical priority of 0 is higher than a task with a numerical priority of 255. Nucleus C++ executes higher priority tasks before lower priority tasks. Tasks having the same priority are executed in the order in which they become ready for execution. A task's priority is defined during task creation. Additionally, dynamic modification of a task's priority is supported.

Preemption

Preemption is the act of suspending a lower priority task when a high priority task becomes ready. For example, suppose a task with a priority of 100 is executing. If an interrupt occurs that readies a task with a priority of 20, the task with a priority of 20 is executed before the interrupted task is resumed. Preemption also occurs when a lower priority task calls a Nucleus C++ service that makes a higher priority task ready. Preemption may be disabled on an individual task basis. When preemption is disabled, no other task is allowed to run until the executing task suspends, relinquishes control or enables preemption. A task that suspends or relinquishes control with preemption disabled has the preemption disabled when it is resumed. A task is created with preemption either enabled or disabled. Preemption may also be enabled and disabled during task execution.

Relinquish

A mechanism is provided to share the processor with other ready tasks at the same priority level in a round-robust fashion. When a task requests this service, all other ready tasks at the same priority are executed before the originally executing task is resumed.

Time Slicing

Time slicing provides another mechanism to share the processor with tasks having the same priority. A time slice corresponds to the maximum number of timer ticks (timer interrupts) that can occur before all other ready tasks at the same priority level are given a chance to execute. A time-slice behaves like an unsolicited task relinquish. Note that disabling preemption also disables time slicing.



Dynamic Creation

Nucleus C++ tasks are created and deleted dynamically. There is no preset limit on the number of tasks an application may have. Each task requires a control block and a stack. The memory for each element is supplied by the application.

Determinism

Processing time associated with suspending and resuming tasks is constant. It is not affected by the number of application tasks. Additionally, the method in which tasks execute is not only predictable, but also guaranteed. Higher priority, ready tasks execute before lower priority, ready tasks. Ready tasks of the same priority execute in the order they become ready.

Stack Checking

Application tasks may check the amount of memory left on the current stack. This function also keeps track of maximum stack usage. Stack checking may also be enabled inside Nucleus C++ services through a conditional compilation option.

Task Information

Application tasks may obtain a list of active tasks. Detailed information about each task can also be obtained. This information includes the task name, current state, scheduled count, priority, and stack parameters.

Priority Pitfalls

Care must be taken when assigning priorities to applications tasks. If care is not taken, the priorities can cause task starvation and excessive system overhead. The class 'Task' is the main base class of all tasks to be defined in the system. It provides an execution object that will be derived from to satisfy the tasking needs specific to the particular application.



Class Structure

The following diagram shows the structure of class `Task`.



Structure of class Task



Creating Your Own Task, 1-2-3

Creating your special task behavior in Nucleus C++ is very easy. Simply derive from the Nucleus C++ PLUS class `Task` and provide an implementation of the `Entry` member. Class `Task` is an abstraction of the mechanisms required to receive a thread of execution from the underlying RTOS. You can operate on your object using class `Task` members.

Subclass class `Task`

The first step to specializing the thread, is to subclass the base class `Task`. The following example specifies a task with a two-kilobyte stack, an initial priority of ten, and an initial time-slice of one second.

The specific subclass contains data elements that consist of an integer (`data`) and an operand (`operand`) that is continuously added to the integer during task execution. Each instance of `MyTask` that is created will have its own local copy of data elements.

The class also contains a pointer to an *ostream* object (`cout_Npp`) to output data on. This stream is created and maintained outside the class and passed in during construction.

```
#include "npp.h"
#include "nppUTIL\nppSTR.h" // simple streams

class MyTask : public Task
{
protected:
    // thread local data.
    int data, operand;
    ostream_Npp* cout_Npp;
public:
    // constructor.
    MyTask( ostream_Npp* init_cout_Npp, const CHAR* name,
            int init_operand)

        // base class parameters.
        : Task( name, 2048 /*stacksize*/, 10 /*priority*/,
              TicksFromMs(1000) /*timeslice*/, TRUE /*preemption*/),

        // data member initializers.
        cout_Npp(init_cout_Npp), data(0), operand(init_operand)

        // empty constructor behavior.
        {}

        // destructor.
        virtual ~MyTask() {}
        // data access.
        virtual int GetData() {return(data);}
}
```



```
protected:
    // data access.
    virtual void SetData(int new_data){data=new_data;}
    // high level data modification.
    virtual void ModifyData() {SetData(GetData()+operand);}
    // high level data presentation.
    virtual void OutputData()
    {
        *cout_Npp << endl << info.GetName()
                << " data: " << GetData()
                << flush;
    }

    // base class callback for receiving the thread.
    virtual void Entry();
};
```

The base class `Task` creates a Nucleus PLUS task service in its constructor and removes the real-time task service in its destructor. The data and operations required to manage the Nucleus PLUS task service are handled automatically. The derived class simply adds data and behavior that is specific to the particular task extension.

Define Task Behavior

The `Entry` member is the callback where the derived class receives the thread of execution from Nucleus PLUS. The following example loops forever. It continuously modifies the data, sleeps for a second, and outputs the data to the demonstration console.

After the call to the `Start` member, the `Entry` member is called by the base class with the *this* pointer properly setup to point to the specific task instance (and therefore its copy of instance data). A base class member is automatically called on the thread of execution that was created in the Nucleus PLUS scheduler and associated with the particular task instance's *this* pointer.

```
void
MyTask::Entry()
{
    info.Update(); // to update the 'name' once for output

    BOOL loop_forever = TRUE;
    while( loop_forever )
    {
        ModifyData();
        Sleep( TicksFromMs(1000) );
        OutputData();
    }
}
```



Create Task Objects

The final step is to create some task objects and start them! Each task that we create and start will have a separate thread running on the underlying Nucleus PLUS kernel. The base class `Task` constructor creates the task service, but does not start it. This separate operation is performed after initialization using the `Start` member.

When the demo executes, you will see the two tasks output their name and the value of their copy of the data (thread local data). The output will appear on the Nucleus C++ demonstration console.

```
void
NppCreate( void* /*first_available_memory*/ )
{
    // Empty, initialize on a task thread.
}

void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS();
    nppPLUS->Initialize();

    // Create a Nucleus C++ standard STREAMS object.
    CreateNppSTDSTREAMS()->Initialize();
    ostream_Npp* cout_Npp =
        NppStandardStreamsComponent()->cout();

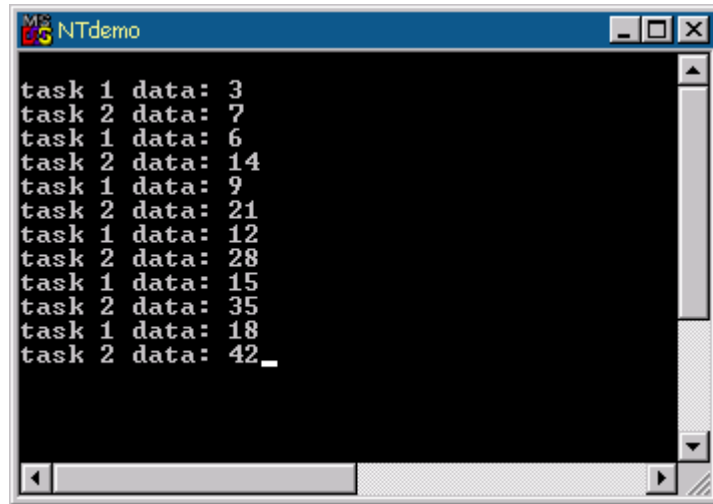
    // Initialize static objects.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    // Create them.
    MyTask* myTask1 = new MyTask( cout_Npp, "task 1", 3 );
    MyTask* myTask2 = new MyTask( cout_Npp, "task 2", 7 );

    // Start them.
    myTask1->Start();
    myTask2->Start();
}
```



The following figure shows the output of the resulting application. Notice `task 1` and `task 2` operate on their own copy of data. Task 1 is incrementing its copy by three each time through the loop and `task 2` is incrementing its copy by seven (as specified during creation on the previous page).



```
task 1 data: 3
task 2 data: 7
task 1 data: 6
task 2 data: 14
task 1 data: 9
task 2 data: 21
task 1 data: 12
task 2 data: 28
task 1 data: 15
task 2 data: 35
task 1 data: 18
task 2 data: 42
```

MyTask Application Output

Both tasks are executing concurrently. The output is synchronized only because the tasks share the same priority and have a long time-slice. Output is completed and each task sleeps before preemption can occur. This side effect is nice, but it is not guaranteed. In the sections to follow, we will overcome this potential design issue using real-time synchronization objects.

Operating on Tasks

Once you have a reference or a pointer to an instance of a task object, you can operate on it to control the thread of execution associated with it.

Be careful! Some members of class `Task` operate on the calling thread, not the thread associated with the implied `this` pointer. Members that operate on the calling thread of execution have static linkage. They are invoked without an instance of class `Task` using the class scope resolution operator (i.e., `Task::Sleep` as opposed to `atask.Sleep`).



Members that Operate on *this* Thread

The members described under this heading operate on the thread associated with the task instance's *this* pointer, not *necessarily* the calling thread. For example, the following changes the priority of the task pointed to by 'that', not the calling thread's priority:

```
void
MyClass::MyMember( Task* that )
{
    // Change the priority of the task pointed to be 'that'.
    OPTION old_priority = that->ChangePriority(10);
}
```

Above, we say not *necessarily* because if the member is being invoked either directly or indirectly from within the Entry member of a class Task subclass, the calling thread is also the thread associated with the implicit *this* pointer:

```
AnotherTask that;

void
MyTask::MyMember()
{
    // Change the priority of this task. This is equivalent
    // to calling this->ChangePriority(10);
    OPTION old_priority = ChangePriority(10);

    // Change the priority of that task to 7. The thread
    // associated with this task remains at priority 10.
    old_priority = that.ChangePriority(7);
}

void
MyTask::Entry()
{
    while(1)
    {
        MyMember();
    }
}
```

To change the priority of a task, use `ChangePriority`. To change the time-slice, use `ChangeTimeSlice`. To determine if a given task is the current running task in the Nucleus PLUS scheduler, use `IsCurrent`. `Reset` resets a task and `Resume` resumes it (`Start` is the same as `Resume`). `Suspend` suspends the specified task and `Terminate` terminates it.

Please see the Class Reference section for details on how to use specific members.



Members that Operate on the caller's Thread

The members described under this heading operate on the calling thread of execution, regardless of whether or not the member is being invoked through a pointer to an instance of a class `Task` subclass. For example, the `Sleep` member operates on the caller's thread. The following shows different calls to `Sleep` and all will cause the calling thread to delay:

```
void
MyClass::MyMember( Task* task )
{
    // WARNING! This sleep call operates on the calling thread,
    // not the thread associated with the task pointed to by
    // 'task'. This will cause a 150 millisecond delay for the
    // calling thread, although syntactically we want to believe
    // the thread associated with 'task' will delay.
    task->Sleep(TicksFromMs(150));

    // This is also the same call:
    Task::Sleep(TicksFromMs(150));
}
```

This is also true for members of class `Task` subclasses. For example, the following member will not cause the thread associated with the task instance to delay, rather it will always cause the calling thread to delay:

```
void
MyTask::MyTaskMember()
{
    // WARNING! This line of code will cause a 150 millisecond
    // delay for the calling thread, not this thread.
    Sleep(TicksFromMs(150));

    // So is this one, the this pointer does not change
    // the behavior of the call to the static member Sleep.
    this->Sleep(TicksFromMs(150));

    // This is also the same call:
    Task::Sleep(TicksFromMs(150));
}
```



This concept is useful for general-purpose routines that require real-time services but are not necessarily “task objects” and therefore have no *tie* into the kernel:

```
void
MyNonTaskClass::SomeMember()
{
    // I need to turn preemption off for the calling thread.
    OPTION old_preemption = Task::PreemptionOff();

    // I need to do a critical chore.
    DoSomething();

    // Restore old preemption posture.
    if( old_preemption == NU_PREEMPT )
    {
        Task::PreemptionOn();
    }
}
```

To turn preemption on and off for the calling thread, use `PreemptionOn` and `PreemptionOff`. To check its stack, use `CheckStack`. To get a pointer to the class `Task` instance associated with the calling thread, use `Current`. To relinquish execution to other threads that share the same priority as the calling thread, use `Relinquish`. To delay for a given number of system ticks on the calling thread, use `Sleep`.

Accessing Task Information

Each class `Task` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `TaskInfo` object that has been filled with updated information about the specific task instance.

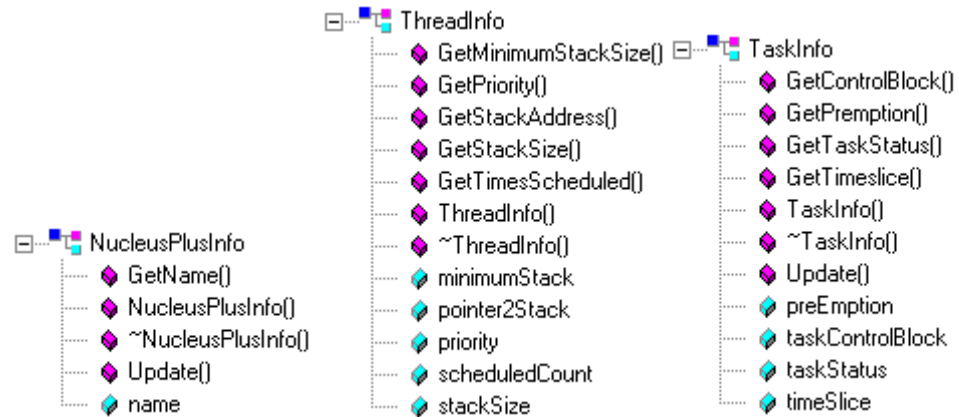
```
// Get a reference to an updated info object from the task.
const TaskInfo& info = myTask.UpdateInfo();

// Do some things with the information.
OPTION priority = info.GetPriority();
UNSIGNED min_stack_size = info.GetMinimumStackSize();
if( info.GetTaskStatus() == NU_FINISHED )
{
    // The task is finished.
}
```



Once the task info object is retrieved, you can simply invoke the members of the `TaskInfo` object to retrieve the following information: the control block, the preemption posture, the task status, the current time-slice and priority, the minimum stack space the task has ever reached, the task's stack address and size, and how many times the task has been scheduled by the Nucleus PLUS kernel.

The following shows the structure of the class `TaskInfo`, a subclass of `ThreadInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class TaskInfo

Please see the *Class Reference* section of this manual for details on how to use specific members.

Inter-task Communication

In Nucleus PLUS, mailboxes, queues, and pipes are very similar. The differences are in the ‘depth’ of the communication channel and the message size. The following highlights the main differences:

Mechanism	Description
Mailbox	Mailboxes provide a low-overhead mechanism to transmit simple messages. Each mailbox is capable of holding a single message the size of four 32-bit words.
Queue	Queues provide a mechanism to transmit multiple messages. A queue message consists of one or more 32-bit words.
Pipe	Pipes provide a mechanism for transmitting multiple messages. A pipe message consists of one or more bytes.

We will focus on class `Q` under this heading of the user’s guide. The use of class `Box` and class `Pipe` parallel this discussion.

Class Q Overview

Queues provide a mechanism for transmitting multiple messages. Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue. Messages may be placed at the front of the queue or at the back of the queue.

Message Size

A message consists of one or more 32-bit words. Both fixed and variable-length messages are supported. The type of message format is defined when the queue is created. Variable-length message queues require an additional 32-bit word of overhead for each message in the queue. Additionally, receive message requests on variable-length message queues specify the maximum message size, while the same requests on fixed-length message queues specify the exact message size.

Suspension

Send and receive queue services provide options for unconditional suspension, suspension with a time-out, and no suspension. Task objects may suspend on a queue for several reasons. A task attempting to receive a message from an empty queue can suspend. Additionally, a task attempting to send a message to a full queue can suspend. A suspended task is resumed when the queue is able to satisfy that task’s request. For example, suppose a task is suspended on a queue waiting to receive a message. When a message is sent to the queue, the suspended task is resumed.



Multiple tasks may suspend on a single queue. Tasks are suspended in either FIFO or priority order, depending on how the queue was created. If the queue supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the queue supports priority suspension, tasks are resumed from high priority to low priority. If a Nucleus C++ member has a suspension option, this option is set to `Q_NU_SUSPEND` by default. This means that if the parameter is absent, tasks are suspended if the request cannot be satisfied.

Broadcast

A queue message may be broadcast. This service is similar to a send request, except that all Task objects waiting for a message from the queue are given the broadcast message.

Dynamic Creation

Nucleus C++ queues are created and deleted dynamically. There is no preset limit on the number of Q objects an application may have.

Determinism

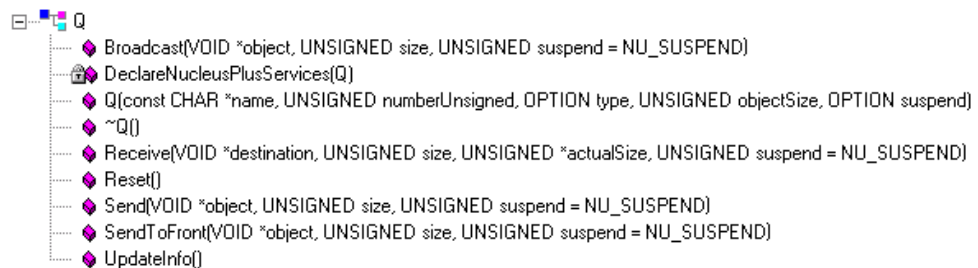
Basic processing time required for sending and receiving queue messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of Task objects currently suspended on the queue.

Queue Information

Application tasks may obtain a list of active queues. Detailed information about each queue can also be obtained. This information includes the queue name, message format, suspension type, number of messages present, and the first task waiting.

Structure of Class Q

The following diagram shows the structure of class Q.



Structure of class Q



Understanding Class Q Behavior

It is important to understand the behavior of Nucleus C++ queues. There are a number of ways to use the classes and how they are used will affect the application. As with most software there are tradeoffs and understanding the application dependent tradeoffs is essential before proceeding to implementation.

Class Q is a Copy-Oriented Queue

Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue (a bit-wise copy of the source into the destination). To improve performance, Nucleus PLUS optimizes the copy when a task is waiting to receive a message (essentially, the message is copied directly to the receiver's destination buffer and bypasses the queue).

Be careful, this copy-oriented behavior may not be the desired one. In C++, an object must be created and destroyed using a constructor and destructor in order to manage the lifetime of its data members that need to be managed by the object. For example, a class may have a string data member that is dynamically created. A bit-wise copy creates two references to the same string! Destroying the second copy is disastrous.

When using Nucleus C++ PLUS queues, it usually makes more sense to send pointers to the objects. Not only will this be quicker (only a pointer is copied), it prevents potential side effects of a bit-wise copy. Another preferred method is to use some form of object serialization.

Most object-oriented languages support the concept of *serialization*. This is the preferred method to *stream* an object into a buffer and vice-versa. Nucleus C++ queues can be used very easily as serialization targets. The sender streams an object into the queue and the receiver streams it out of the queue.

Class Q is a Type-Less Container

Nucleus C++ queues are built on the foundation of the Nucleus PLUS queue service that provides essential real-time capabilities. However, the Nucleus PLUS queue service is type-less. It is up to the user of the queue service to determine and manage message sizes and types.

Since the main goal of all Nucleus C++ components is to provide a thin abstraction of Nucleus "C" services, the Nucleus C++ Q class encapsulates the fundamental capabilities of the Nucleus PLUS queue service. Thus, members take VOID pointers and message sizes. While it is OK to use the class as is (without deriving from it), in C++ we prefer the concept of *types*.

Class Q Subclasses are Typed Containers

Because we have a thin abstraction and a need for typed messages, Nucleus C++ PLUS provides macros to implement (pre C++ template) typed queues. These macros simply build a class declaration and definition based on the *type macro parameter* you provide.



NOTE: It is possible and somewhat better to implement typed containers using C++ templates. However, to maximize portability we chose to use macros since not all Nucleus C++ targeted C++ implementations support templates. We need to guarantee Nucleus C++ applications remain portable. C++ template versions are available in the files `NPPTEMPL.H`, `NPPTEMPL.INL`, and `NPPTEMPL.CPP`.

Using Queues, 1-2-3

Using queues in Nucleus C++ is very easy. The following example steps the reader through the steps necessary and also shows a variety of ways that two Nucleus PLUS threads can communicate and share data.

Declare the Message Type

For this example, we will create an Apple class and a simple subclass of it, GreenApple. This will be the type of message we will send between two Nucleus PLUS threads. One thread will contain and produce apples and the other will consume them.

```
// Simple apple class.
class Apple
{
    public:
        // Simple const/dest sets name to apple.
        Apple() : bites(0) {strcpy(name,"apple");}
        virtual ~Apple() {}

        // Data accessors.
        UNSIGNED NumberBites() const {return(bites);}
        const CHAR* Name() const {return(name);}

        // virtual routine to count bites.
        virtual void Bite() {bites++;}

    protected:
        // Simple data for each instance.
        UNSIGNED bites;
        char name[16];
};

// Simple apple subclass.
class GreenApple : public Apple
{
    public:
        // Simple const/dest sets name to green_apple.
        GreenApple() {strcpy(name,"green_apple");}
        virtual ~GreenApple() {}

        // Green apples are harder to get down.
        // Each bite takes ten bites.
        virtual void Bite() {bites+=10;}
};
```



The constructors of the classes set the name data member so we can distinguish between and identify the types of messages, not necessarily to identify individual instances.

The other piece of data in the class is a counter, `Bites`, used to keep track of how many times the apple has been bitten. There is a virtual member, `Bite`, that is called to bite the apple. The `GreenApple` subclass overrides the behavior of the `Bite` member in the base class to bite the apple ten times for every bite request.

This simple demonstration of polymorphism is here to demonstrate we can communicate data between tasks using OO techniques when sending pointers but we cannot do this by sending the object itself directly without an object serialization method. This also reinforces our understanding that Nucleus PLUS queues are copy-oriented.

Declare and Define the Typed Apple Queue Class

Now that we have a specific type of message we want to send and receive, let's build the class that communicates apples. We will demonstrate two simple ways to communicate apples, one communicates by sending a pointer to an apple and the other sends a copy of the apple object.

First, declare the class using the typed queue macros:

```
// Declare a queue of Apple that will cause tasks to
// suspend on a FIFO basis. The class is AppleFifoQ.
DeclareQ( Apple, Fifo )

// Declare a queue of Apple* that will cause tasks to
// suspend on a FIFO basis. The class is ApplePtrFifoQ.
typedef Apple* ApplePtr;
DeclareQ( ApplePtr, Fifo )
```

The first macro builds a class declaration of a simple class `Q` subclass that provides overrides of the base class members to copy (bit-wise) the `Apple` instance into and out of the queue. The resulting class name is `AppleFifoQ` (tasks will suspend on the queue in first-in-first-out order).

The second macro does the same thing, only it builds a class declaration of a simple class `Q` subclass that provides overrides of the base class members to copy only a pointer to the `Apple` instance into and out of the queue. The resulting class name is `ApplePtrFifoQ`. Notice we needed to use a `typedef` to define a single type that is a pointer to an `Apple` instance since the macros are literal text substitutions.

We now have a derived Nucleus C++ PLUS queue class that is typed to send apples. If you try to use a member of the class and pass in an orange, you will get a compile time error. We have regained the strong type-checking features of C++.



We cannot forget to define the class! The following shows how to define the class. More than likely, this will be put in an implementation file and the declaration macros described above will be put in a header file. This maximizes reuse:

```
// Define the queues.
DefineQ( ApplePtr, Fifo )
DefineQ( Apple, Fifo )
```

Declaring Tasks that Communicate to Each Other

Now we need to implement a couple of simple tasks that communicate apples. The following class declarations define a quarterback and a receiver. Both the quarterback and the receiver use a shared external queue. Both take pointers to the queues in the constructor and maintain the link with pointer data members.

We do this for simplicity to demonstrate how the queues work. You might want to include the communication queue inside the receiver and expose communication members that have better context to the receiver.

The class `Receiver` declaration is shown below. Notice we do not have any message types declared within the class although we will be receiving apple messages from the quarterback. Where is the message data for the receiver? To further our understanding of tasks, the receiver will instantiate the apple data local on each instance's stack (as local parameters to the `Entry` member).

```
// A simple task that receives messages.
class Receiver : public Task
{
public:
    // Simple const/dest sets up queue pointers
    // and creates/deletes a task service.
    Receiver
    (
        ApplePtrFifoQ* init_ptr_queue,
        AppleFifoQ*      init_queue
    )
    :
    Task( "Receiver"/*name*/, 1536 /*stacksize*/,
        10 /*priority*/, 5 /*timeslice*/,
        TRUE /*preemption*/ ),
    ptr_queue( init_ptr_queue ), queue( init_queue )
    { /*empty*/ }
    virtual ~Receiver() { /*empty*/ };
};
```



```
protected:
    // Queues we receive messages from.
    ApplePtrFifoQ* ptr_queue;
    AppleFifoQ* queue;

    // Receiving task entry member. We will continuously
    // loop receiving messages.
    virtual void Entry();
};
```

The class Quarterback declaration is shown below. This class is very similar to the receiver except that this class declares the message data within the class itself. The quarterback has two instances of apples, one regular apple and a green apple.

```
// A simple task that sends messages.
class Quarterback : public Task
{
public:
    // Simple const/dest sets up queue pointers
    // and creates/deletes a task service.
    Quarterback
    (
        ApplePtrFifoQ* init_ptr_queue,
        AppleFifoQ* init_queue
    )
    :
    Task( "QB"/*name*/, 1536 /*stacksize*/,
        10 /*priority*/, 5 /*timeslice*/,
        TRUE /*preemption*/ ),
    ptr_queue( init_ptr_queue ), queue( init_queue )
    { /*empty*/ }
    virtual ~Quarterback() { /*empty*/ };

protected:
    // Queues we send messages to.
    ApplePtrFifoQ* ptr_queue;
    AppleFifoQ* queue;

    // The apples we are sending.
    Apple apple;
    GreenApple green_apple;

    // Sending task entry member. We will continuously
    // loop sending messages.
    virtual void Entry();
};
```



Creating the Application Task Instances

To create the quarterback and receiver, simply create the objects at startup. The following example shows the code that creates the apple pointer queue that holds twenty apple pointer messages and the apple queue that holds ten apple messages.

```
void
NppCreate( void* /*first_available_memory*/ )
{
    // empty.
}

void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS();
    nppPLUS->Initialize();

    // Nucleus C++ STREAMS component.
    CreateNppSTDSTREAMS()->Initialize();

    // Nucleus C++ STATIC component.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    // Create message queues.
    ApplePtrFifoQ* ptr_queue = new ApplePtrFifoQ("appleptr",20);
    AppleFifoQ* queue = new AppleFifoQ("apple",10);

    // Create the tasks.
    Quarterback* quarterback = new Quarterback(ptr_queue,queue);
    Receiver* receiver = new Receiver(ptr_queue,queue);

    // Start the tasks.
    quarterback->Start();
    receiver->Start();
}
```



Define the Behavior of the Quarterback Task

The quarterback continuously loops, sending its apples to both queues. First, it sends its apple to the apple queue. Then it sends both a pointer to the apple and a pointer to its green apple to the apple pointer queue. Each apple is bitten at the end of each loop.

```
void
Quarterback::Entry()
{
    // Setup pointer references once.
    Apple* apple_ptr = &apple;
    Apple* green_apple_ptr = &green_apple;

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Send the apple.
        queue->Send( apple );

        // Send both apple pointers.
        ptr_queue->Send( apple_ptr );
        ptr_queue->Send( green_apple_ptr );

        // Bite our apples to update our data.
        apple.Bite();
        green_apple.Bite();
    }
}
```

Notice we cannot send the subclass instance, `green_apple`, to the apple queue because its type is wrong. Trying to send a green apple will not make it through the compiler due to strong type checking. This is good since the apple queue only copies enough data for the base class. This is the advantage of pointer queues. We can send subclass instances and take full advantage of polymorphism. The apple pointer queue is heterogeneous and the apple queue is homogenous.

You will notice there is nothing that will cause this task to delay except for queue full conditions. This is implied by the default arguments to the `Send` member of the queue class. The default parameter is `NU_SUSPEND`. This means that if the queue is full, the task will conditionally suspend until there is room in the queue for the message.



Alternatively, a timeout specification could have been provided. The following code snippet shows the same call to the `Send` member using a few different timeout specifications. To check the results, examine the status return value. The possible return values are documented in the reference chapter for class `Q`.

```
STATUS status;

// This call conditionally suspends if the queue is full.
status = queue->Send( apple );

// This call also conditionally suspends if the queue is full.
status = queue->Send( apple, NU_SUSPEND );

// This call does not suspend at all if the queue is full.
status = queue->Send( apple, NU_NO_SUSPEND );

// This call conditionally suspends for 100 milliseconds or
// until the queue can hold a message, whichever is first.
status = queue->Send( apple, TicksFromMs( 100 ) );
```

Define the Behavior of the Receiver Task

The receiver continuously loops, receiving its apples from both queues. First, it receives an apple from the apple queue. Then it receives both a pointer to an apple and a pointer to a green apple from the apple pointer queue.

As the apples are received, the receiver prints each apple's information to the demo console. As mentioned above, the apple instance and the pointer to the apple that gets filled in with the message contents is located on the task's stack.

```
void
Receiver::Entry()
{
    // The apple object and pointer we are receiving.
    Apple* apple_ptr;
    Apple apple;

    // Use Nucleus C++ Singleton cout for all demo output.
    ostream_Npp& cout_Npp
        = *NppStandardStreamsComponent()->cout();

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Receive the apple and print the data.
        queue->Receive( apple );
        cout_Npp << endl << "bites for " << apple.Name()
            << ": " << apple.NumberBites();

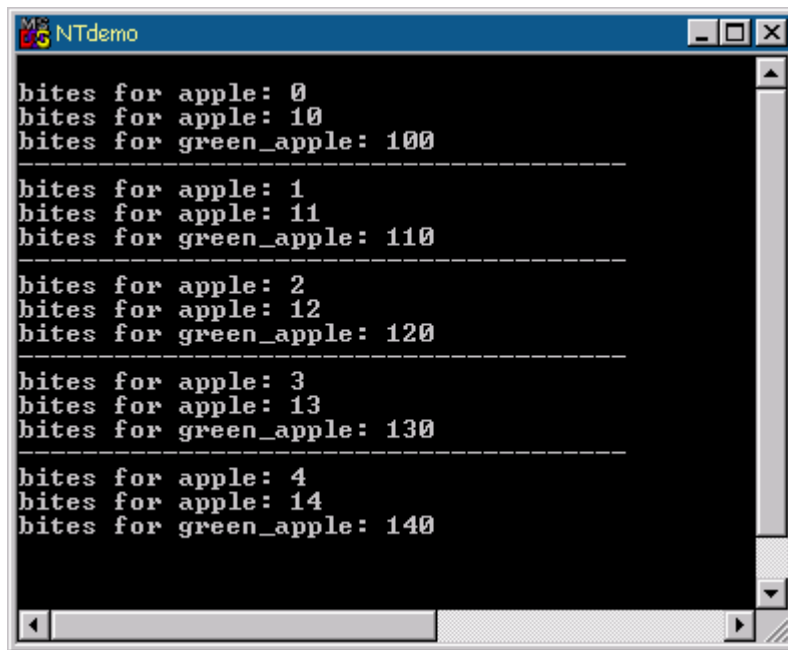
        // Receive both apple pointers and print the data.
        for( int i=0; i<2; i++ )
        {
```



```
ptr_queue->Receive( apple_ptr );
cout_Npp << endl << "bites for "
        << apple_ptr->Name()
        << ": " << apple_ptr->NumberBites();
}

// Output banner line and sleep for a half-second.
cout_Npp << endl << HalfBannerLine;
Sleep( TicksFromMs( 500 ) );
}
```

Each time through the loop, the receiver sleeps for a half-second. This is for ease of human reading and also causes the sender to suspend as the queues become full. The figure below shows the output of the resulting application.



```
NTdemo
bites for apple: 0
bites for apple: 10
bites for green_apple: 100
-----
bites for apple: 1
bites for apple: 11
bites for green_apple: 110
-----
bites for apple: 2
bites for apple: 12
bites for green_apple: 120
-----
bites for apple: 3
bites for apple: 13
bites for green_apple: 130
-----
bites for apple: 4
bites for apple: 14
bites for green_apple: 140
```

Output of the class Q Demo Application



What is the Difference Between the Queues?

By examining the output of the application, you will notice that the receiver is printing out different results (the number of bites) for each apple as they are being received. Why? Why is the green apple showing so many more bites? Why is the number of bites reported for the regular apple ten bites different in each case, regardless of the fact the sender sent the same apple, at the same time?

Copy-Oriented Queues

The reason the regular apple is reporting discrepancy of a ten bites is due to the nature of the queue that was used to send the message. In the case of sending the apple to the apple queue, the apple was *copied* into the queue. Thus, what was actually sent was the apple data at a snapshot in time.

In the case of sending the apple to the apple pointer queue, all that was copied into the queue was the pointer (the message *type*). Thus, once the receiver receives the message (during its time slice since both the quarterback and the receiver are at the same priority), the data has changed. The sender was biting the apple while the receiver was not *running*.

This is a very *simple* example used to demonstrate the nature of the Nucleus C++ PLUS inter-task communication classes. The operation of a bit-wise copy of a C++ class is not always the best thing to do. Again, if we do want this type of behavior, please investigate *object serialization design patterns* for assistance in this area.

Also, to keep this application simple, we ignored the fact that the sender and receiver may operate on the apple at the same time. The access to this shared data should be thread protected using a Nucleus C++ PLUS inter-task synchronization object. These details are discussed below.

Heterogeneous Queues and Polymorphism

The reason the green apple is showing so many more bites is due to polymorphism. The green apple type overrides the `Bite` member and bites it ten times for each bite request. The receiver was able to access the data resulting from this extended behavior since the pointer it received can be a pointer to a base class `Apple` or any subclass of it. This is the essence of Liskov's Substitution Principle. The Nucleus C++ PLUS queue, although type-less, does not limit this feature of the C++ language.



Using Class Q Directly

As mentioned above, the base class `Q` can be used directly, it is not an abstract base class. You may want to use the class directly to meet specific needs like object serialization. In this case, it is up to the user of the class to cast type names and determine the size of the messages. This is not difficult, just a bit confusing.

The best way to understand how to use the class without the assistance of the type macros is to examine (and copy) the implementation of the macros themselves. They demonstrate how to use the `sizeof` operator to achieve this task.

Accessing Queue Information

Each class `Q` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `QInfo` object that has been filled with updated information about the specific queue instance.

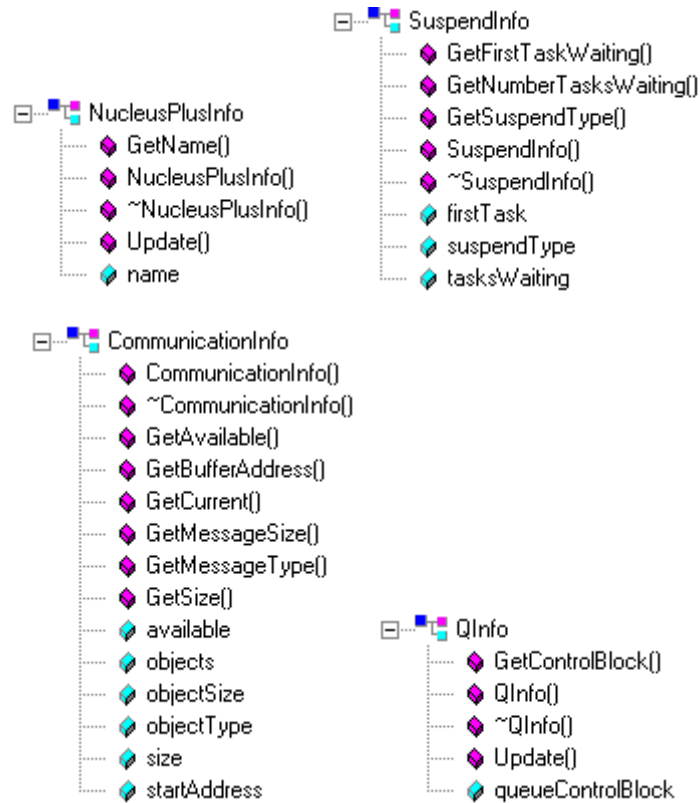
```
// Get a reference to an updated info object from the queue.
const QInfo& info = myQueue.UpdateInfo();

// Do some things with the information.
UNSIGNED available = info.GetAvailable();
UNSIGNED current = info.GetCurrent();
UNSIGNED task_waiting = info.GetNumberTasksWaiting();
```

Once the queue info object is retrieved, you can simply invoke the members of the `QInfo` object to retrieve the following information: the control block, the total number of `UNSIGNED` data elements, the number of available `UNSIGNED` data elements in the queue, the number of messages currently in the queue, the number of tasks waiting on the queue, and other information relating to the instance of queue the information object was obtained from.



The following shows the structure of the class `QInfo`, a subclass of `CommunicationInfo`, a subclass of `SuspendInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class `QInfo`

Please see the Class Reference chapter for details on how to use specific members.



Periodic Timers

Using Nucleus PLUS timers, the Nucleus C++ PLUS class `Timer` provides a C++ class interface for supporting real-time periodic timers. Programmers either create new timer expiration behaviors or reuse existing behaviors that solve related duties.

Class `Timer` Overview

Nucleus C++ provides the application with programmable timer objects. These timers execute a specific derived object member routine when they expire. The object's expiration routine executes as a high level interrupt service routine. Therefore, self suspension requests are not allowed. Additionally, processing should be kept to a minimum. A `Timer` object provides a mechanism to execute a timer routine when they expire. The routine executes as a high-level interrupt service routine. Therefore, self-suspension requests are not allowed. Processing should be kept to a minimum in derived `Timer` object expiration routines.

Re-Scheduling

When a timer expires, the prescribed expiration member routine of the derived object is executed. After execution is complete, the timer is either dormant or rescheduled. If the timer's reschedule value is zero, it is dormant after the initial expiration. However, if the timer's rescheduled value is non-zero, it is rescheduled to expire at that interval.

Enable/Disable

Application timers may be automatically enabled during creation. Additionally, timers may be enabled and disabled dynamically.

Reset

The initial ticks, rescheduling rate, and the expiration routine of a timer may be reset dynamically by the application.

Dynamic Creation

Nucleus C++ application timers are created and deleted dynamically. There is no preset limit on the number of `Timer` objects an application may have.



Determinism

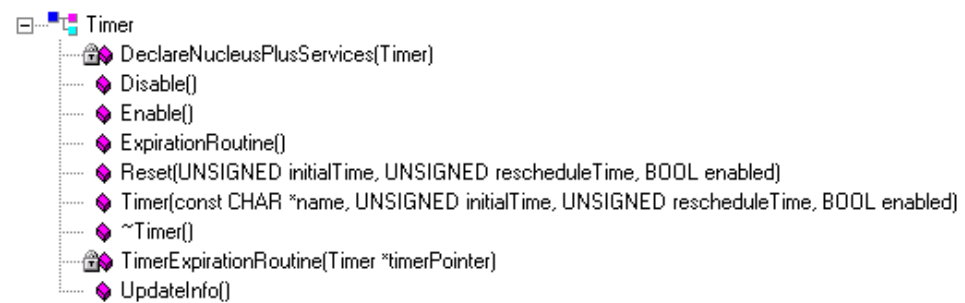
Processing time required to create, enable, disable, and modify application timers is constant. However, processing time required to execute the user-supplied expiration routines depends on the expiration routines themselves, and the number of timers that expire simultaneously.

Information

Application tasks may obtain a list of active timers. Detailed information about each timer is also available. This information includes the timer name, status, initial ticks, reschedule value, remaining ticks, and the expiration count.

Structure of Class Timer

The following diagram shows the structure of class `Timer`.



Structure of class Timer

Using Timers, 1-2-3

Using timers in Nucleus C++ is very easy. Simply derive from the Nucleus C++ PLUS class `Timer` and provide an implementation of the `ExpirationRoutine` member. Class `Timer` is an abstraction of the mechanisms required to receive a timer thread callback from the underlying RTOS. You can operate on your object using class `Timer` members.



Continuing from the Previous Example

The example in this section continues from our previous queue example. To simplify the application for this discussion, we want to make some simple changes. We will modify the receiver and quarterback tasks to only use the apple pointer queue. The resulting task code is shown below.

The difference is that both the quarterback and the receiver simply use a queue and a single apple data variable.

```
class Receiver : public Task
{
public:
    // Simple const/dest sets up queue pointers
    // and creates/deletes a task service.
    Receiver( ApplePtrFifoQ* init_queue )
    : Task("Receiver"/*name*/,1536/*stacksize*/,
        10/*priority*/,5/*timeslice*/,
        TRUE/*preemption*/),
      queue(init_queue)
    { /*empty*/ }
    virtual ~Receiver() { /*empty*/ };

protected:
    // Our queue and apple.
    ApplePtrFifoQ* queue;
    Apple*         apple;

    // Receiving task entry member. We will continuously
    // loop receiving messages.
    virtual void Entry();
};

class Quarterback : public Task
{
public:
    // Simple const/dest sets up queue pointers
    // and creates/deletes a task service.
    Quarterback( ApplePtrFifoQ* init_queue )
    : Task("QB"/*name*/,1536/*stacksize*/,
        10/*priority*/,5/*timeslice*/,
        TRUE /*preemption*/),
      queue( init_queue ), apple("qb_apple")
    { /*empty*/ }
    virtual ~Quarterback() { /*empty*/ };

protected:
    // Our queue and apple.
    ApplePtrFifoQ* queue;
    Apple         apple;

    // Sending task entry member.
    virtual void Entry();
};
```



The change to the task behaviors is shown below. The quarterback simply continuously loops, sending its apple to the queue. The receiver simply continuously loops, receiving its apple from the queue, printing its data to the console, and sleeping for a half-second.

The only thing that will cause the quarterback thread to suspend is queue full conditions.

```
void
Quarterback::Entry()
{
    // Setup pointer references once.
    Apple* apple_ptr = &apple;

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Send the apple.
        queue->Send( apple_ptr );
    }
}

void
Receiver::Entry()
{
    // Use Nucleus C++ Singleton cout for all demo output.
    ostream_Npp& cout_Npp
        = *NppStandardStreamsComponent()->cout();

    cout_Npp << endl << HalfBannerLine << endl
        << "Receiver started and waiting."
        << endl << HalfBannerLine << endl;

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Receive the apple and print the data.
        queue->Receive( apple );
        cout_Npp << endl << "bites for " << apple->Name()
            << ": " << apple->NumberBites();

        // Sleep for a half-second.
        Sleep( TicksFromMs( 500 ) );
    }
}
```



Subclass class Timer

The first step to specializing the timer is to subclass the base class `Timer`. The following example specifies a timer that will expire once for every period specified by the `ms` millisecond parameter passed into the constructor.

The specific subclass contains data elements that consist of a green apple and a pointer to the queue to send the apple to. Each instance of `QuarterbackTimer` that is created will have its own local copy of data elements.

```
class QuarterbackTimer : public Timer
{
public:
    // Simple const/dest sets up queue and creates/deletes
    // a timer service to expire every "ms" milliseconds.
    QuarterbackTimer( ApplePtrFifoQ* init_q, UNSIGNED ms )
    :   Timer("qbtimer", TicksFromMs(ms)/*initial time*/,
        TicksFromMs(ms)/*reschedule time*/,
        FALSE/*initially disabled*/),
        queue(init_q), green_apple("timer_apple")
    { /*empty*/ }
    virtual ~QuarterbackTimer() { /*empty*/ };

protected:
    // Our queue and apple.
    ApplePtrFifoQ* queue;
    GreenApple     green_apple;

    // Expiration routine for the timer.
    virtual void ExpirationRoutine();
};
```

The base class `Timer` creates a Nucleus PLUS timer service in its constructor and removes the real-time timer service in its destructor. The data and operations required to manage the Nucleus PLUS timer service are handled automatically. The derived class simply adds data and behavior that is specific to the particular timer extension.



Define Timer Expiration Behavior

The `ExpirationRoutine` member is the callback where the derived class receives the timer thread of execution from Nucleus PLUS. The following example simply sends the apple to the queue every time the timer expires.

The `ExpirationRoutine` member is called by the base class with the *this pointer* properly setup to point to the specific timer instance (and therefore its copy of instance data). A base class member is automatically called on the timer thread of execution that was created in the Nucleus PLUS scheduler and associated with the particular timer instance's *this pointer*.

```
void
QuarterbackTimer::ExpirationRoutine()
{
    // Simply send our green apple. Notice we cannot
    // suspend if the queue is full.
    Apple* green_apple_ptr = &green_apple;
    queue->Send( green_apple_ptr, NU_NO_SUSPEND );
}
```

You will notice we cannot suspend if the queue is full because timers execute as high-level interrupt service routines. Therefore, self-suspension requests are not allowed and processing should be kept to a minimum.



Create Objects

The final step is to create the task and timer objects and start them! The base class `Timer` constructor creates the timer service with a flag that determines if the timer is automatically enabled or not. This example does not automatically start the timer in order for us to demonstrate the `Enable` operation on the timer (the `Disable` member is used in the same way). This separate operation to enable the timer is performed after initialization.

```
void
NppCreate( void* /*first_available_memory*/ )
{
    // Empty, initialize on a task thread.
}

void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS();
    nppPLUS->Initialize();

    // Create a Nucleus C++ standard STREAMS object.
    CreateNppSTDSTREAMS()->Initialize();
    ostream_Npp* cout_Npp =
        NppStandardStreamsComponent()->cout();

    // Initialize static objects.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    // Create message queue.
    ApplePtrFifoQ* queue = new ApplePtrFifoQ("apple",10);

    // Create and start the receiver.
    Receiver* receiver = new Receiver(queue);
    receiver->Start();
    // Create and start the quarterback.
    Quarterback* quarterback = new Quarterback(queue);
    //COMMENTED_OUT quarterback->Start();

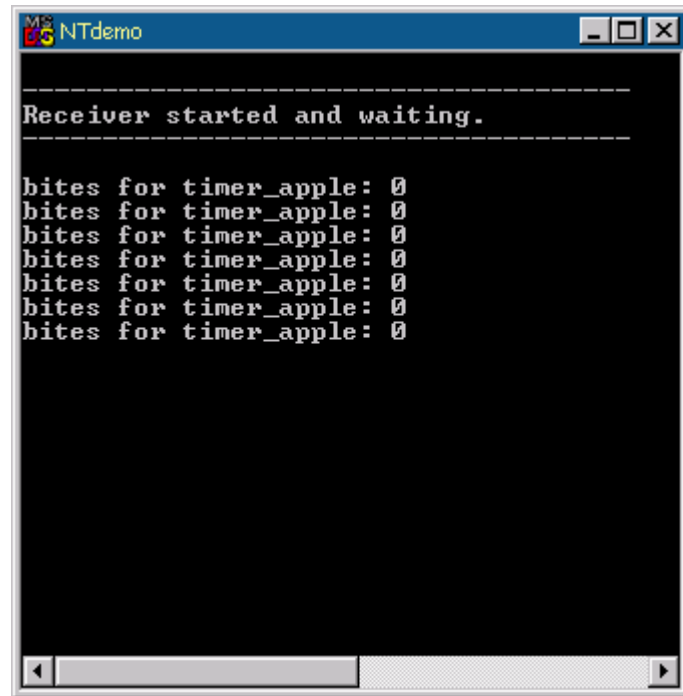
    // Create the timer to expire every 5 seconds.
    QuarterbackTimer* timer = new QuarterbackTimer(queue,5000 );

    // Enable the timer. Note the Timer base class constructor
    // takes a flag to specify the initial enable state.
    timer->Enable();
}
```



You will notice in the above example that we have commented out the call to start the quarterback task. This is done in this step in order to demonstrate the timer. In the next step, we will start the quarterback task and examine its effect.

The output of this demonstration application is shown below. We are not concerned with the data since we are not eating the apples (no calls to the `Bite` method). In this incarnation of the example, we only have the timer and the receiver running since we have the call to start the quarterback commented out.



```
NTdemo
-----
Receiver started and waiting.
-----
bites for timer_apple: 0
bites for timer_apple: 0
bites for timer_apple: 0
bites for timer_apple: 0
bites for timer_apple: 0
bites for timer_apple: 0
bites for timer_apple: 0
```

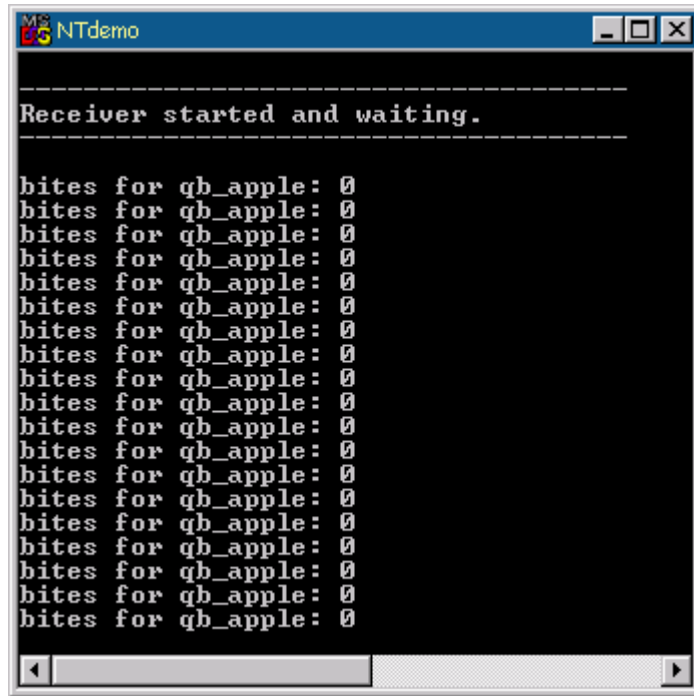
Timer Application Output

The timer expires every five seconds and sends the apple message to the queue that the receiver is suspending on. The receiver receives the message and prints the apple data. The apple data indicates the message came from the timer.



Start the Quarterback

This time, we will start the quarterback task and examine the results. You will notice the timer messages are never getting through, although setting a breakpoint in the timer expiration routine shows the timer is firing. All we get is the quarterback apple messages. What is happening?



```
Receiver started and waiting.

bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
```

Timer Application Output with QB Task Running

The reason the timer messages are not going through is the queue is filling up. You will notice the call to the `Send` member of the queue class is being parameterized to specify that the caller is not suspended if the queue is full:

```
queue->Send( green_apple_ptr, NU_NO_SUSPEND );
```

The reason for this is we cannot suspend inside a high-level interrupt service routine and that is what the timer executes as. Trying to do so will result in a run-time error. We need to rectify this problem in the application. We need a mechanism for the timer to communicate to the quarterback thread telling it to back off and let others play in this game.

We will do this with another form of inter-task communication, *events*. The next section will pick up from here and fix the issue, allowing both the quarterback task and the quarterback timer to send messages to the receiver.



Accessing Timer Information

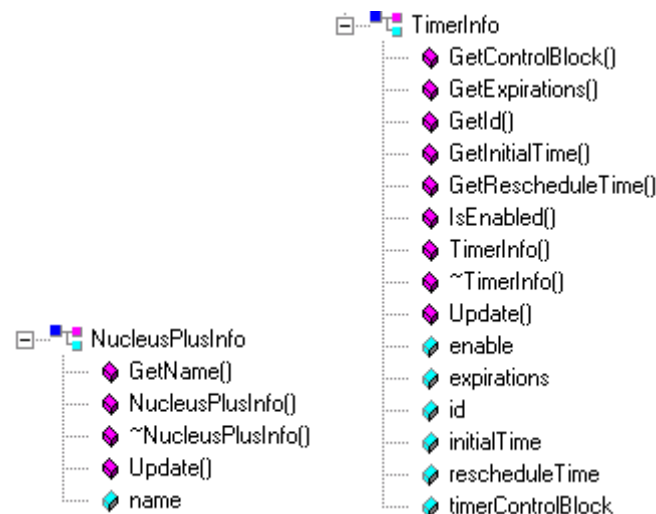
Each class `Timer` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `TimerInfo` object that has been filled with updated information about the specific timer instance.

```
// Get a reference to an updated info object from the timer.
const TimerInfo& info = myTimer.UpdateInfo();

// Do some things with the information.
UNSIGNED expirations = info.GetExpirations();
UNSIGNED reschedule_time = info.GetRescheduleTime();
BOOL enabled = info.GetEnabled();
```

Once the `TimerInfo` object is retrieved, you can simply invoke the members of the `TimerInfo` object to retrieve the following information: the control block, the number of times the timer has expired, its initial and reschedule tick durations, and whether or not the timer is enabled.

The following shows the structure of the class `TimerInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class `TimerInfo`

Please see the Class Reference chapter for details on how to use specific members.



Events

The Nucleus C++ PLUS event classes are used to create real-time architectures that are event-driven, providing a commonly required synchronization mechanism. The Nucleus C++ classes are `Event` and `EventGroup`. Event groups support the ability to suspend waiting for a logical combination of multiple events to occur.

Class `EventGroup` Overview

Event groups provide a mechanism to indicate that a certain system event has occurred. An event is represented by a single bit in an event group. This bit is called an event flag. There are 32 event flags in each event group. Event flags can be set and cleared using logical AND/OR combinations. Event flags can be received in logical AND/OR combinations as well. Additionally, event flags may be reset automatically after they are received.

Suspension

The `receive` event flag requests provide options for unconditional suspension, suspension with a time-out, and no suspension. A task attempting to receive a combination of event flags that are not present can suspend. Resumption of the task occurs when a `set` event-flags operation satisfies the combination of events requested by the task.

Multiple tasks may suspend trying to receive different combinations of event flags from the same event group. All tasks suspended on an event group are checked for resumption when a `set-event-flags` operation is performed on the event group.

All Nucleus C++ members that provide suspension options have their `suspend` parameter set to `NU_SUSPEND` by default. This means that if the parameter is absent (or set to `NU_SUSPEND`), tasks are suspended if the request cannot be satisfied.

Dynamic Creation

Nucleus C++ event groups are created and deleted dynamically. There is no present limit on the number of event groups an application may have.

Determinism

Processing time required for receiving event flags from an event group is constant. However, the processing time required to set event flags in an event group is affected by the number of tasks suspended in the event group.

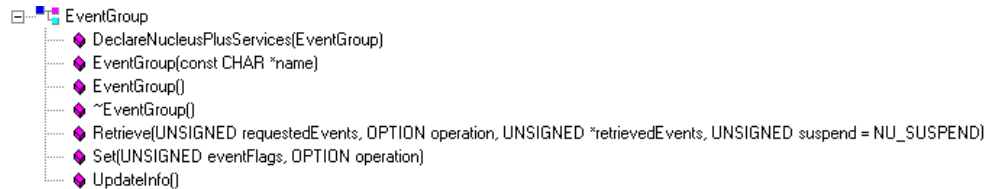


Information

Application tasks may obtain a list of active event groups. Detailed information about each event group is also available. This information includes the event group name, current event flags, number of tasks waiting, and the first task waiting.

Structure of Class EventGroup

The following diagram shows the structure of class EventGroup.



Structure of class EventGroup

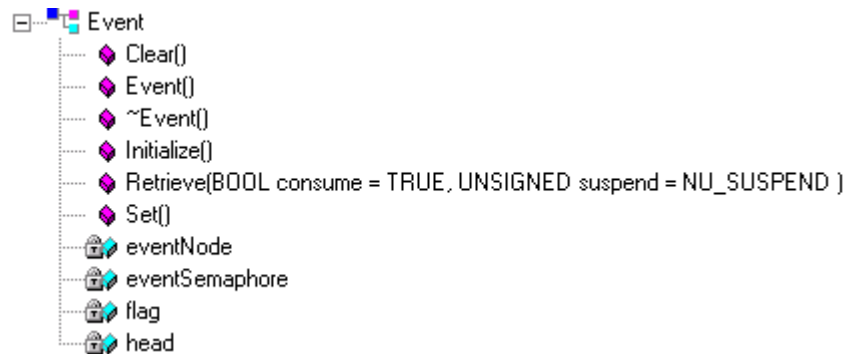
Class Event Overview

The class Event provides a mechanism to indicate that a certain event has occurred. An event may be set, queried, and automatically cleared after it is sensed. The class has been implemented using the Nucleus PLUS real-time event group service. The details required to manage bit masks and the underlying event group service are handled by the class. Event group services are reused to create subsequent event instances.

The class EventGroup is used to manage multiple events that a thread is interested in and class Event is used to manage a single isolated event.

Structure of Class Event

The following diagram shows the structure of class Event.



Structure of class Event



Using Events, 1-2-3

Using events in Nucleus C++ is very easy. Simply create an event object and use its members to operate on it.

Continuing from the Previous Example

The example in this section continues from our previous timer example. We left the application with the problem that it would not allow the timer quarterback to send messages to the receiver since the quarterback task was filling up the queue. Since timers cannot suspend, we did not have the luxury to block until the queue had room for the message.

Therefore, we must implement a mechanism to communicate from the timer thread to the task thread. We will do this by setting an event in the quarterback task that tells it to back off, stop sending messages, and block (conditionally suspend) until further notified. The task will be further notified through the same event.

Modify the Quarterback Classes

We need to modify the quarterback task and the quarterback timer to add the event. We will put the event object in the quarterback task since the task is what needs to be notified (by potentially multiple sources). Then, we will provide an access member to get at it and pass that to the timer to use.

The following shows the modifications necessary for the quarterback timer class. Notice we have added a pointer to an event data member and initialized it by the event pointer that is passed into the constructor.

```
class QuarterbackTimer : public Timer
{
public:
    // Simple const/dest sets up queue and creates/deletes
    // a timer service to expire once every "ms"
    // milliseconds.
    QuarterbackTimer(ApplePtrFifoQ* init_q, UNSIGNED ms,
        Event* init_event )
    :   Timer("qbtimer", TicksFromMs(ms)/*initial time*/,
        TicksFromMs(ms)/*reschedule time*/,
        FALSE/*initially disabled*/),
        queue(init_q), green_apple("timer_apple"),
        event(init_event)
    { /*empty*/ }
    virtual ~QuarterbackTimer() { /*empty*/ };

protected:
    // Our queue and apple.
    ApplePtrFifoQ* queue;
    GreenApple     green_apple;
    Event*         event;

    // Expiration routine for the timer.
    virtual void ExpirationRoutine();
};
```



The following shows the modifications necessary for the quarterback task class. Notice we have added an event data member and added access to it with the `GetEvent` member.

```
class Quarterback : public Task
{
public:
    // Simple const/dest sets up queue pointers
    // and creates/deletes a task service.
    Quarterback( ApplePtrFifoQ* init_queue )
    : Task("QB"/*name*/,1536 /*stacksize*/,10/*priority*/,
        5/*timeslice*/,TRUE/*preemption*/),
      queue( init_queue ), apple("qb_apple")
    { /*empty*/ }
    virtual ~Quarterback() { /*empty*/ };

    Event* GetEvent() { return(&event); }

protected:
    // Our queue and apple.
    ApplePtrFifoQ* queue;
    Apple          apple;
    Event          event;

    // Sending task entry member.
    virtual void Entry();
};
```

Modify the Quarterback Task Behavior

Lets examine the new quarterback task behavior. The following shows the new `Entry` routine for the task.

```
void
Quarterback::Entry()
{
    // Setup pointer references once.
    Apple* apple_ptr = &apple;

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Send the apple.
        queue->Send( apple_ptr );

        // Check to see if we hogged the queue.
        STATUS status = event.Retrieve( TRUE, NU_NO_SUSPEND );
        if( status == NU_SUCCESS )
        {
            // We got in the way. Wait until the event is
            // set again, indicating that the timer was
            // able to send a message.
            event.Retrieve( TRUE, NU_SUSPEND );
        }
    }
}
```



Added are checks for whether the `event` data member has been set. Notice the task is just checking when it performs the first event retrieval. It does not suspend if it isn't set. This is specified by the `NU_NO_SUSPEND` parameter value.

If it is successful retrieving the event (meaning the event has been set by the timer), the event is consumed. This means the event is automatically cleared. This is specified by `TRUE` parameter passed into the `Retrieve` member. Alternatively, if this parameter value is `FALSE`, the event remains set.

The results of the first operation are indicated by the return value. If it is set (indicated by `NU_SUCCESS`), it proceeds into the `if` statement and the thread blocks on the same event data member until it has been set again.

The difference in the second event retrieval is the second one specifies that the thread will suspend until the event has been set. We do this not only to demonstrate a second method of retrieving an event, but this is the way the timer can notify the thread that it was able to get a message through. After this, the quarterback task can resume and fill the queue again. This process is repeated continuously.

Modify the Quarterback Timer Behavior

Lets examine the new quarterback timer behavior. The following shows the new `ExpirationRoutine` member for the timer.

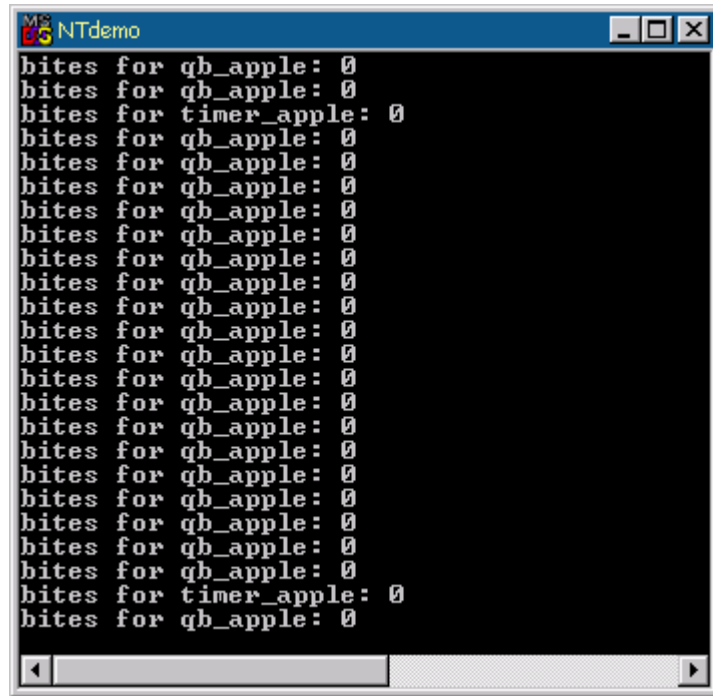
```
void
QuarterbackTimer::ExpirationRoutine()
{
    // Simply send our green apple. Notice we cannot
    // suspend if the queue is full.
    Apple* green_apple_ptr = &green_apple;
    STATUS status = queue->Send(green_apple_ptr, NU_NO_SUSPEND);
    if( status != NU_SUCCESS )
    {
        // The queue is full, set the event to
        // tell the quarterback to back off.
        event->Set();
    }
    else
    {
        // We were able to send the message. Check to see if
        // the event is set. If it is not, we need to tell
        // the quarterback that it is ok to start again.
        STATUS status = event->Retrieve(FALSE, NU_NO_SUSPEND);
        if( status != NU_SUCCESS )
        {
            // The event isn't set. Set it to start the QB.
            event->Set();
        }
    }
}
```

If the queue `Send` operation is not successful (indicated by the lack of a successful return value), the timer sets the quarterback task's event telling it to stop sending messages until further notified by this timer.



If it was successful sending the message, it checks to see if the event is clear by trying to retrieve it and looking for a non-success return value. This indicates to the timer that it must notify the task to resume by setting the event again. You will notice the timer specifies to not consume the event.

This inter-thread communication synchronizes the shared access to the shared resource. The resulting application output is shown below. Notice now that timer messages are getting through.

A screenshot of a Windows NTdemo application window. The window has a blue title bar with the text "NTdemo" and standard Windows window controls (minimize, maximize, close). The main area of the window is black with white text. The text consists of a list of messages, each on a new line. The messages are: "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for timer_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for qb_apple: 0", "bites for timer_apple: 0", and "bites for qb_apple: 0". The messages are arranged in a vertical list, with the "bites for timer_apple: 0" messages appearing at the third and second-to-last positions in the list. The window has a scroll bar at the bottom right, indicating that the list of messages can be scrolled through.

```
bites for qb_apple: 0
bites for qb_apple: 0
bites for timer_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for qb_apple: 0
bites for timer_apple: 0
bites for qb_apple: 0
```

New Timer Application Output



Events and Suspension

The suspension characteristics of class `Event` parallel all other Nucleus C++ PLUS suspension options. A timeout specification can be provided. The following code snippet shows calls to the `Retrieve` member using a few different timeout specifications. To check the results, examine the status return value. The possible return values are documented in the reference chapter for class `Event`.

```
STATUS status;

// This call conditionally suspends if the event is clear. The
// default is to consume the event automatically.
status = event->Retrieve();

// This call also conditionally suspends if the event is clear.
status = event->Retrieve ( TRUE, NU_SUSPEND );

// This call does not suspend at all if the event is clear.
status = event->Retrieve ( TRUE, NU_NO_SUSPEND );

// This call conditionally suspends for 100 milliseconds or
// until the event is set, whichever is first.
status = event->Retrieve ( TRUE, TicksFromMs( 100 ) );
```

Using Event Groups, 1-2-3

Using event groups in Nucleus C++ is easy. You simply need to manage the concept of a bit mask representing a group of events. Each bit identifies one of the events. This gives the programmer the ability to act on a number of events and check the event flag states in some logical combination.

Declare Some Bit Meanings

The following example shows a simple button panel application. Each button is assigned a bit in the bit mask.

```
#define POWER_BUTTON    0x00000001
#define NEXT_BUTTON     0x00000002
#define SELECT_BUTTON   0x00000004
```



Do Something and Set the Event Flags

We will assume a memory mapped I/O register that we read from. The value read from the register will parallel our bit definitions (mighty convenient!). All we need to do is call the `Set` method with the current state of the button bit mask. The `Set` operation is parameterized to specify a logical OR operation. In this way, we will not ‘*erase*’ old bits.

```
// a memory mapped I/O register
#define BUTTON_REG (volatile (UNSIGNED*) 0x1FFF1234)

void
ButtonHardwareHighlevelInterrupt::Entry()
{
    UNSIGNED button_reg;

    // the user has pressed a button, check which
    // one(s) and set the bits in the button event
    // group accordingly. I assume that I am reading
    // a button "register" from memory mapped I/O
    button_reg = *(BUTTON_REG);

    // we have a one to one correspondence
    // between the button register bits and the
    // masks defined above. We specify to do a
    // bitwise OR of the current event group
    // bitmask with the button_reg value that
    // we just read in.
    pEvents->Set( button_reg, NU_OR );
}
```



Check the Status of the Event Flags

Now let's assume a task is looking for either the 'next' or 'select' button. All we need to do is retrieve the event flags and examine them. We can choose to suspend until both are set, or either one is set. The following example shows the "either-one" method. Other possible logical combinations are possible. Please refer to the Class Reference chapter on class `EventGroup` for more information.

```
void UserInterfaceTask::Entry()
{
    STATUS    status;
    UNSIGNED  actual_mask;

    while( 1 )
    {
        // Suspend up to 1000 milliseconds waiting for either the
        // next or select buttons to be pressed.
        status = pEvents->Retrieve((NEXT_BUTTON|SELECT_BUTTON),
                                   NU_OR_CONSUME/*operation*/, &actual_mask,
                                   TicksFromMs(1000));

        if(status == NU_SUCCESS)
        {
            // We got a button press event! Check which one(s).
            if(actual_mask & NEXT_BUTTON) HandleNextButton();
            if(actual_mask & SELECT_BUTTON) HandleSelectButton();
        }
    }
}
```

Accessing EventGroup Information

Each class `EventGroup` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `EventGroupInfo` object that has been filled with updated information about the specific event group instance.

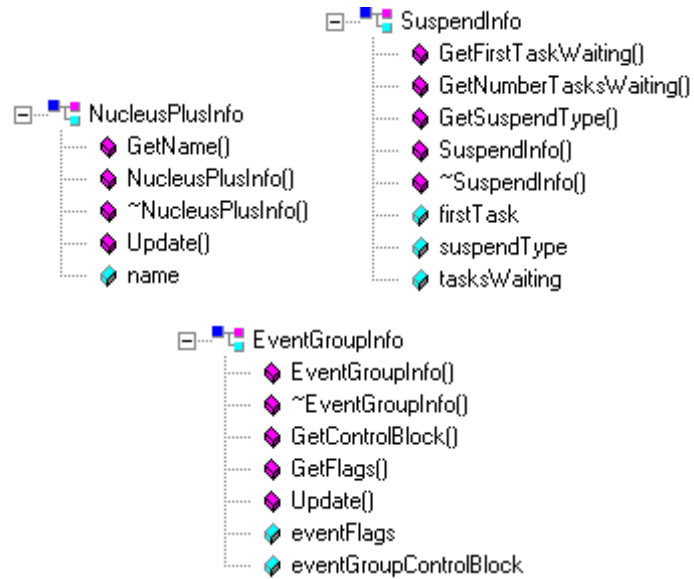
```
// Get a reference to an updated info object from the event group.
const EventGroupInfo& info = myEventGroup.UpdateInfo();

// Do some things with the information.
UNSIGNED flags = info.GetFlags();
UNSIGNED number_tasks_waiting = info.GetNumberTasksWaiting();
```

Once the event group info object is retrieved, you can simply invoke the members of the `EventGroupInfo` object to retrieve the following information: the control block; the current event flag contents; the number of tasks waiting on the event group; etc.



The following shows the structure of the class `EventGroupInfo`, a subclass of `SuspendInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class `EventGroupInfo`

Please see the [Class Reference](#) section for details on how to use specific members.

Semaphores

The Nucleus C++ PLUS classes `Semaphore`, `Priority Semaphore` and `FifoSemaphore` are used to manage multithreaded access to critical shared resources. These critical section management classes are implemented using Nucleus PLUS semaphore services.

Class Semaphore Overview

The Nucleus C++ PLUS classes `Semaphore`, `Priority Semaphore`, and `Fifo Semaphore` are used to manage multithreaded access to critical shared resources. These critical section management classes are implemented using Nucleus PLUS semaphore services.

Semaphores provide a mechanism to control execution of critical sections of an application. Nucleus C++ provides counting semaphores that range in value from 0 to 4,294,967,294. The two basic operations on a semaphore are `obtain` and `release`. Obtain-semaphore requests decrement the semaphore, while release-semaphore requests increment the semaphore. Semaphore objects provide a mechanism to control execution of critical sections of an application. Nucleus C++ provides counting semaphores that range in value from 0 to 4,294,967,294. The two basic operations on a semaphore are `obtain` and `release`. Obtain semaphore requests decrement the semaphore, while release semaphore requests increment the semaphore. Resource allocation is the most common application of a semaphore. Additionally, semaphores created with an initial value can be used to indicate an event.

Suspension

The obtain-semaphore service provides options for unconditional, suspension, suspension with a time-out, and no suspension. A task attempting to obtain a semaphore whose count is currently zero can suspend. Resumption of the task is possible when a release-semaphore request is made. Multiple tasks may suspend trying to obtain a single semaphore. Tasks are suspended in either FIFO or priority order, depending on how the semaphore was created. If the semaphore supports FIFO suspension, tasks are resumed in the order in which they tried to obtain the semaphore. Otherwise, if the semaphore supports priority suspension, tasks are resumed from high priority to low priority. All Nucleus C++ members that provide suspension options have their `suspend` parameter set to `NU_SUSPEND` by default. This means that if the parameter is absent or the (parameter is `NU_SUSPEND`), tasks are suspended if the request cannot be satisfied.



Priority Inversion

Priority inversion occurs when a higher priority task is suspended on a semaphore that a lower priority task has. This situation is unavoidable if different priority tasks share the same protected resources. In such situations, a limited and predictable amount of time in priority inversion is acceptable. However, if the low priority task is preempted by a middle priority task during a priority inversion situation, the amount of time in priority inversion is no longer deterministic. Such a situation can be avoided by insuring that all tasks using the same semaphore have the same priority, at least while they own the semaphore.

Dynamic Creation

Nucleus C++ semaphores are created and deleted dynamically. There is no preset limit on the number of semaphores an application may have. Semaphores may be created with any initial count.

Determinism

Processing time required for obtaining and releasing semaphores is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the semaphore.

Information

Application tasks can obtain a list of active semaphores. Detailed information about each semaphore is also available. This information includes the semaphore name, current count, suspension type, number of tasks waiting, and the first task waiting.

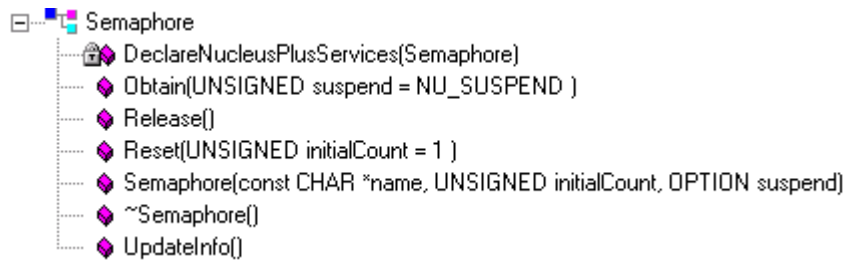
Deadlock

A deadlock refers to a situation where two or more tasks are forever suspended attempting to obtain two or more semaphores. The simplest example of the situation is a system with two tasks and two semaphores. Suppose the first task has the second semaphore and the second task has the first semaphore. Now suppose that the second task attempts to obtain the second semaphore and the first task attempts to obtain the first semaphore. Since each task has the semaphore that the other needs, the tasks could suspend on the semaphore forever. Prevention is the preferred way to deal with deadlocks. This technique imposes rules on how semaphores are used by the application. For example, if tasks are not allowed to possess more than one semaphore at a time, deadlocks are prevented. Alternatively, deadlocks may be prevented if tasks obtain multiple semaphores in the same order. The optional time-out on obtain-semaphore suspension can be used to recover from a deadlock situation.



Structure of Class Semaphore

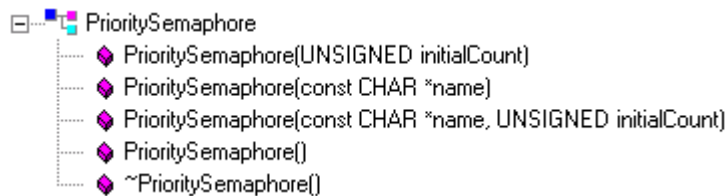
The following diagram shows the structure of class Semaphore.



Structure of class Semaphore

Structure of Class PrioritySemaphore

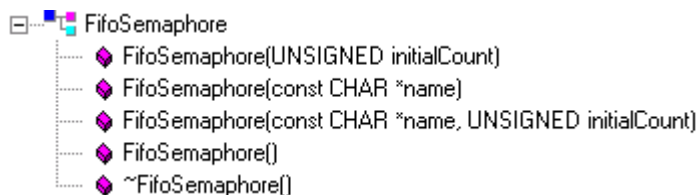
The following diagram shows the structure of class PrioritySemaphore, a simple subclass of class Semaphore.



Structure of class PrioritySemaphore

Structure of Class FifoSemaphore

The following diagram shows the structure of class FifoSemaphore, a simple subclass of class Semaphore.



Structure of class FifoSemaphore



What is a Semaphore?

A semaphore is a fundamental synchronization mechanism for multithreaded operating systems used to synchronize access to shared resources among multiple threads that are competing for access to that shared resource.

The Problem

Before we discuss the solution, let's discuss the problem. Semaphores are used in a variety of situations where critical sections inside a multi-threaded application are present. A critical section is an area of code where the number of threads that 'enter' that area, or section, needs to be controlled.

The simplest of these situations is demonstrated by our simple queue demonstration application. It is possible that both the quarterback and the receiver will access the memory location associated with the `Byte` data member at the same time.

For example, the quarterback might be in the middle of the `Byte` member, writing to the data to update its value. During this operation, the quarterback thread can be preempted by the RTOS and the receiver proceeds to read the data value from the shared object. This could be a very bad thing if the *writer* was only $\frac{1}{2}$ of the way through the write and the *reader* reads the value as a partially updated value.

There are many other problems in the discipline of multithreaded software development that semaphores solve. For example: they can be used to indicate an event has occurred; they can be used to control access to shared resources outside of memory (for example, a serial device); they can be used to communicate between threads (another means of inter-task communication); etc.

The Solution

We need a synchronization mechanism to synchronize access to the shared data element. This is the main purpose of class `Semaphore`. A semaphore is an object that can be '*obtained*' by a thread of execution. Once the semaphore object is obtained, other threads that attempt to obtain the same instance of the semaphore are suspended until the other thread '*releases*' it.



Deciding the Order of Suspension

Just like the inter-task communication services, the semaphore services in Nucleus PLUS allow the programmer to specify how tasks suspend (and eventually continue) when the service cannot be accommodated at any given instance in time. The programmer can specify, at the time of service creation, whether tasks wake up in first-in-first-out order (FIFO) or they wake up in order of priority. In the case of threads with the same priority, the service defaults back to FIFO.

For ease of use, the Nucleus C++ PLUS component provides three semaphore classes: `Semaphore`, `PrioritySemaphore`, and `FifoSemaphore`. The base class `Semaphore` requires the programmer to pass on construction parameters that tell the class how to instantiate the semaphore service.

The other two classes pass this information to Nucleus PLUS automatically on behalf of the programmer based on the type of semaphore object being created.

`PrioritySemaphore` causes priority suspension and `FifoSemaphore` causes FIFO suspension.

Using Semaphores, 1-2-3

Using semaphores in Nucleus C++ is very easy. This section shows the steps required and discusses alternative ways to use the classes. Continuing from the previous example for class `Q`, let's add critical section protection to the `Bite` data member of class `Apple`.

Add a Semaphore Object to the Class

Adding a semaphore object to a class is as easy as adding any other data member. Just simply put the object in the class:

```
class Apple
{
    // ...
    protected:
        PrioritySemaphore semaphore;
};
```



Use the Semaphore to Protect Shared Data

Now that we have an object, we can use it to synchronize access to the shared `Bite` data element. When we want to access the shared resource, we obtain the critical section by executing the `Obtain` operation of our semaphore class. When we are done with it, we release it using the `Release` operation. The changed members of class `Apple` are:

```
UNSIGNED NumberBites()
{
    UNSIGNED return_value;
    semaphore.Obtain();
    {
        return_value = bites;
    }
    semaphore.Release();
    return(return_value);
}

virtual void Bite()
{
    semaphore.Obtain();
    {
        bites++;
    }
    semaphore.Release();
}
```

In the derived class `GreenApple` we must do the same thing. The derived class has access to the `Bite` data element directly. This is not the best design since we really should be wrapping access up in protected members of the base class but we want to simply demonstrate the principle characteristics of the semaphore.

```
virtual void Bite()
{
    semaphore.Obtain();
    {
        bites+=10;
    }
    semaphore.Release();
}
```

Notice we do not require protection in the member initialization list since the object does not completely exist at the time of object instantiation and therefore the data member cannot be accessed.

You may also notice another subtle change made to the `NumberBites` member. The `const` specification was removed since we are actually operating on the class and the constness of the member no longer holds true. This was done on purpose here to demonstrate that multithreaded access must be planned into classes that will cooperate across multiple threads (which means most of today's needs!).



If we tried to change the class later, we would need to change the signature of a key member and that could break existing dependent classes, especially subclasses. It is essential to design thread synchronization into a class and not try to do it as an afterthought.

Using Suspension Parameters

It may not be desirable to suspend while waiting on access to the critical section in all situations. Because of this, you can pass suspension parameters into the members that allow you to timeout if the section cannot be obtained within a given number of RTOS timer ticks.

The following shows some examples of different suspension possibilities. The status value can be inspected to determine the results and see if the semaphore was actually obtained or not. Possible return values are listed in the reference chapter for the Obtain operation. Please refer to that section for details.

```
STATUS status;

// The following will suspend the calling thread unconditionally
// until the semaphore can be obtained.
status = semaphore.Obtain();

// So will this.
status = semaphore.Obtain( NU_SUSPEND );

// The following will return immediately if the semaphore cannot
// be obtained.
status = semaphore.Obtain ( NU_NO_SUSPEND );

// This call conditionally suspends for 100 milliseconds or
// until the semaphore can be obtained, whichever is first.
status = semaphore.Obtain( TicksFromMs( 100 ) );
```

Using the Counting Features

Nucleus PLUS semaphores are actually counting semaphores. The example above shows what is called a *binary semaphore*, or a semaphore with an initial count of one.

Alternatively, an initial count of some arbitrary number can be sent into the service, either at construction time or by using the Reset operation of class Semaphore. The Nucleus PLUS service manages the semaphore 'count'. Obtain requests decrement the count, while release requests increment the count.

Threads will continue to be allowed into the critical section until the count gets to zero. At that point, the number of simultaneous threads allowed in the section has been exhausted and subsequent threads must suspend until the counter becomes non-zero or they must get out of town.



This is useful in situations when you have a limited number of resources that multiple threads use. For example, your device might have a modem pool that is managed with a counting semaphore. For example, if we have seven modems, we would create a semaphore with an initial count of seven. As threads request use of a modem, they simply attempt to obtain the semaphore. When the seven modems are in use, subsequent obtains fail until another thread releases the semaphore when they are done using their modem.

Accessing Semaphore Information

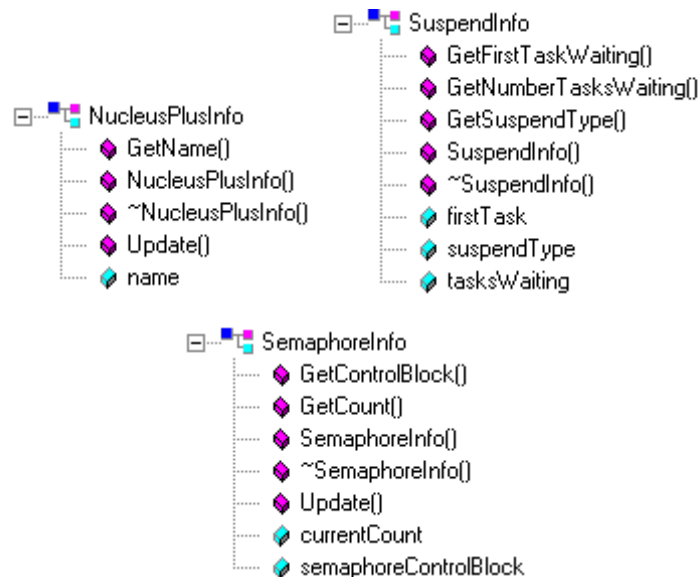
Each class `Semaphore` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `SemaphoreInfo` object that has been filled with updated information about the specific semaphore instance.

```
// Get a reference to an updated info object from the semaphore.
const SemaphoreInfo& info = mySemaphore.UpdateInfo();

// Do some things with the information.
UNSIGNED count = info.GetCount();
UNSIGNED number_tasks_waiting = info.GetNumberTasksWaiting();
```

Once the semaphore info object is retrieved, you can simply invoke the members of the `SemaphoreInfo` object to retrieve the following information: the control block; the current semaphore count; the number of tasks waiting on the semaphore; etc.

The following shows the structure of the class `SemaphoreInfo`, a subclass of `SuspendInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class SemaphoreInfo



Please see the functional reference section for details on how to use specific members.

External Interrupts

The Nucleus C++ PLUS classes `LowLevelInterrupt` and `HighLevelInterrupt` are base classes that manage asynchronous external interrupts in an embedded application. The classes demonstrate the power of portability. The various interrupt architectures found in today's advanced embedded processors are very different, but the Nucleus C++ PLUS class interfaces remain constant across all of them!

Class `LowLevelInterrupt` Overview

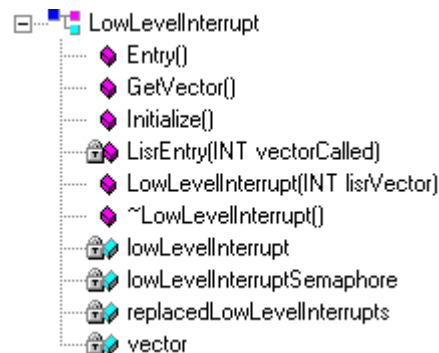
A low level interrupt object provides a mechanism for producing an immediate response to an external or internal event. When an interrupt occurs, the processor suspends the current path of execution and transfers control to an interrupt service routine (ISR) that calls this object's `Entry` member.

The `Entry` member executes as a normal ISR, which means it uses the stack of the current executing thread. Nucleus PLUS saves the thread's context for you. The entire mechanism has been specifically designed for real-time embedded systems and is very efficient.

Subsequent object instantiations for the same interrupt vector will cause old objects to be replaced. When the object that replaces an old object is destructed, the old object is reinstalled. Multiple replacements can occur and old objects will be reinstalled on a last-replaced-first-reinstalled basis.

Structure of Class `LowLevelInterrupt`

The following diagram shows the structure of class `LowLevelInterrupt`.



Structure of class `LowLevelInterrupt`



Class HighLevelInterrupt Overview

The `HighLevelInterrupt` class provides an interface into the Nucleus PLUS kernel's high level interrupt services. Each `HighLevelInterrupt` has its own stack space and its own control block. It can therefore be blocked if it tries to access a kernel data structure that is already being accessed.

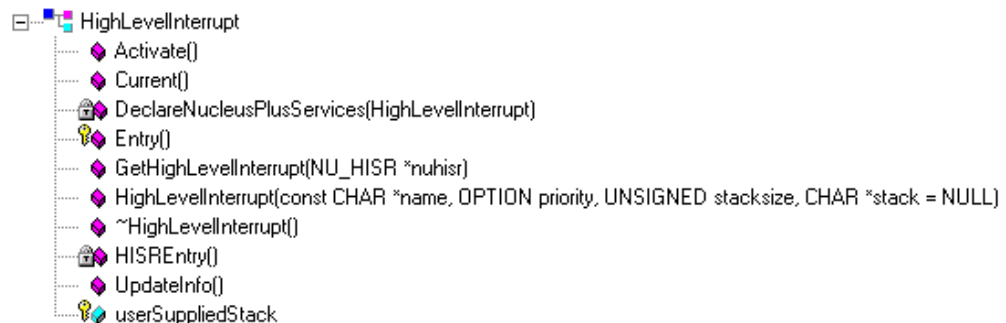
`HighLevelInterrupts` are allowed access to most kernel services, with the exception of self-suspension services. Therefore, a `HighLevelInterrupt` cannot suspend on a kernel service. It must run to completion and return before normal task scheduling can proceed.

There are three priority levels available to `HighLevelInterrupts`. If a higher priority `HighLevelInterrupt` is activated during processing of a lower priority `HighLevelInterrupt`, the lower priority `HighLevelInterrupt` is preempted in much the same manner as a Task gets preempted. `HighLevelInterrupts` of the same priority are executed in the order in which they were originally activated.

An activation counter is maintained for each `HighLevelInterrupt`. It is used to assure the object's `Entry` member is executed once for each activation request.

Structure of Class HighLevelInterrupt

The following diagram shows the structure of class `HighLevelInterrupt`.



Structure of class HighLevelInterrupt

Using Interrupts, 1-2-3

Using interrupts in Nucleus C++ is very easy. The use is very similar to the way tasks and timers are created and used. Simply derive from the Nucleus C++ PLUS interrupt classes and provide an implementation of the virtual `Entry` members.

Class `LowLevelInterrupt` is an abstraction of the mechanisms required to receive a low-level interrupt callback from the underlying RTOS. `HighLevelInterrupt` is an abstraction of the mechanisms required to receive a high-level interrupt callback from the underlying RTOS. You can operate on your objects using class members.

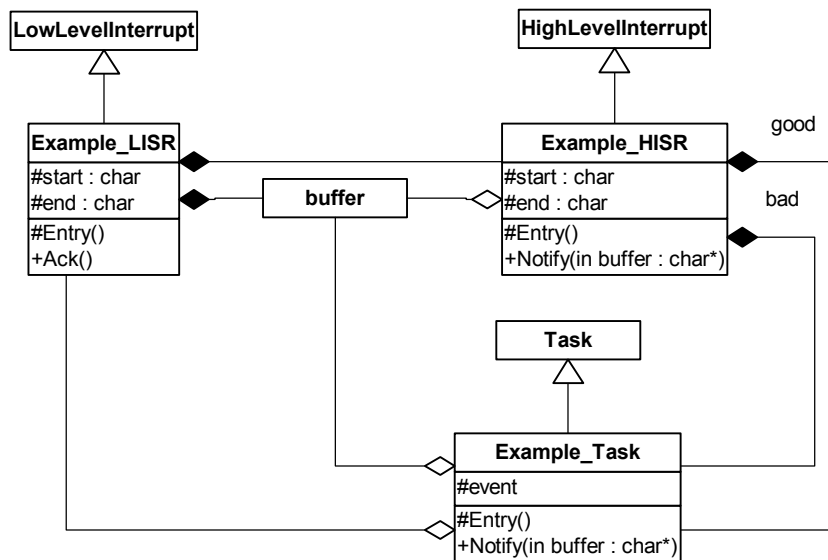


Example Application Overview

The problem is to fill a buffer with characters that are received one-by-one from a source that interrupts every time a new character is available. The buffer needs to be serviced at a relatively lower priority in the overall application and therefore we need to push the handling of the buffer to a task thread where full kernel services, including suspension, are available.

Example Application Class Structures

The example in this section is a bit more complicated than previous example applications in this guide, primarily due to the number of classes that work in conjunction to achieve the final interrupt processing behavior. To simplify the discussion, the following UML diagram presents an overview of the class structures found in the example.



Example Interrupt Application Structure

We do not want to service the buffer on every character, therefore we will split the handling into three layers: low-level interrupt, high-level interrupt, and finally task. This demonstrates three distinct classes of execution threads that are possible in a Nucleus PLUS system. The rest of this section describes the key differences and where each is applicable to embedded real-time systems.



As shown above, we will manage each specific type of execution thread that appears in our example as a subclass of our base Nucleus C++ PLUS classes. We will reuse the base classes to handle the details required to register the interrupt services with the underlying Nucleus PLUS kernel and we will supply behavior in the subclass that is specific to the needs of the example application.

LISR

The packet buffer is contained within the `Example_LISR` composite along with attributes that describe the packet boundaries (start and end characters in this simple example). Also within the composite class is the `Example_HISR` object that handles packets in the high-level interrupt thread.

The `Example_LISR` class provides an override of the virtual callback that is called each time the interrupt associated with the instance's *vector* fires. This callback is the `Entry` method and is called immediately upon receipt of each character. The `Example_LISR` class constructor takes a Nucleus PLUS *vector* number that the object will be installed under. This is the connection between the object instance and the underlying interrupt vector table that is managed and serviced by Nucleus PLUS.

In addition, the `Example_LISR` class provides a virtual method, `Ack`, used to acknowledge the object that the buffer has been handled by the task layer and the buffer can be filled with the next packet.

HISR

The `Example_HISR` class is also a composite that contains the start and end characters describing the packets. It also contains two instances of class `Example_Task`, one that handles good packets and the other to handle bad packets that are received.

The `Example_HISR` class provides an override of the virtual callback that is called each time the high-level interrupt is activated. The activation request is automatically scheduled by the RTOS each time the `Activate` member of the object is invoked. This callback is the `Entry` method and is called once for each invocation of the method.

In addition, the `Example_HISR` class provides a virtual method, `Notify`, used to notify the object that a new buffer is available for processing.

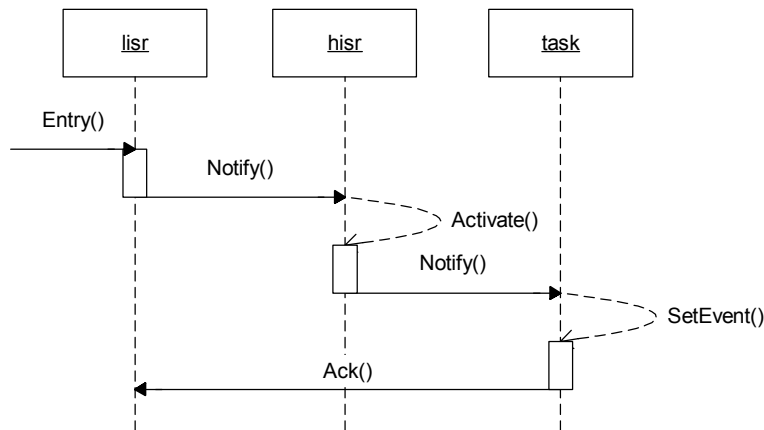
Task

In addition to the normal `Entry` method each task provides, the `Example_Task` class provides a virtual method, `Notify`, used to notify the object that a new buffer is available for processing. The class contains a synchronization event that is used to indicate the new buffer is available. The task blocks on this event until it is set.



Example Application Sequence

The following shows a UML sequence diagram that depicts the message sequencing that appears in the example application as time progresses downward.



Example Interrupt Application Sequence

LISR

The basic idea is that the low-level interrupt (`lisr`) object's `Entry` method is called each time a character becomes available. This is done automatically by the RTOS each time the interrupt vector fires. Since we do not want to schedule a high-level interrupt (`hisr`) and subsequently a task each time a single character is received, we will perform some minor processing inside the `lisr`'s `Entry` member.

Although we do need to perform some processing inside `Entry` in order to achieve our above goal, it is kept to an absolute minimum to balance the tradeoffs that result from the fact that we are executing as a normal processor ISR. This means we are executing on top of the current thread of execution that the processor happened to be executing at the time of interrupt. This can be any type of thread, including any nested conditions that arise.

Therefore, we will simply insert each character into the buffer and only act further if we see a packet boundary character. If it is a start character, it recognizes that we missed a valid packet and starts back at the beginning of the buffer (effectively throwing the packet away). If it sees the end character, it terminates the buffer and passes it along its journey into the high-level interrupt and tasking lands.



HISR

The act of sending the buffer along its way is accomplished by notifying the `histr` object of the event by invoking its `Notify` member. We need to think bare-bones C++ for a moment. When we invoke the `histr`'s `Notify` member, we are still executing on the low-level interrupt thread, on top of the thread context of the interrupted thread. Therefore, processing must again be kept to a minimum within the `Notify` member.

The only processing performed is to activate the `histr` instance by invoking its own `Activate` member (implicitly through the *this* pointer). When the `Notify` member and subsequently the `histr`'s `Entry` member return, the high-level interrupt service is scheduled by Nucleus PLUS and the `histr` object's `Entry` member is automatically called with the *this* pointer properly setup to point to the instance.

Since we do not want to schedule the task if the buffer is invalid, we will perform some minor processing inside the `histr`'s `Entry` member. Although this example probably performs too much processing for a real interrupt architecture, we do this to demonstrate that we have three different classes of processing to handle an interrupt on the three distinct *types* of threads.

Inside the `histr`'s `Entry` member we are executing on a Nucleus PLUS high-level interrupt thread. Thus, the normal rules for Nucleus PLUS high-level interrupt services apply. Most kernel services are available with the exception of self-suspension requests.

We do have the advantage of priority driven scheduling to handle tough interrupt application requirements, but we are limited to not being able to request suspension and we must execute to completion in order for normal task scheduling to resume. Therefore, we will simply check the validity of the packet and act accordingly. Regardless of whether the packet is good or bad, it passes it along its journey into tasking land.

Task

The act of sending the buffer along its way is accomplished by notifying the task object of the event by invoking its `Notify` member. We need to think bare-bones C++ for a moment again. When we invoke the task's `Notify` member, we are still executing on the high-level interrupt thread. Therefore, processing must again be kept to a minimum within the task's `Notify` member.

The only processing performed is to set the event data member within the task. The task is blocking on this event until it is set. When the task's `Notify` member and subsequently the `histr`'s `Entry` member return, the task is re-scheduled by Nucleus PLUS using its normal task scheduling rules.

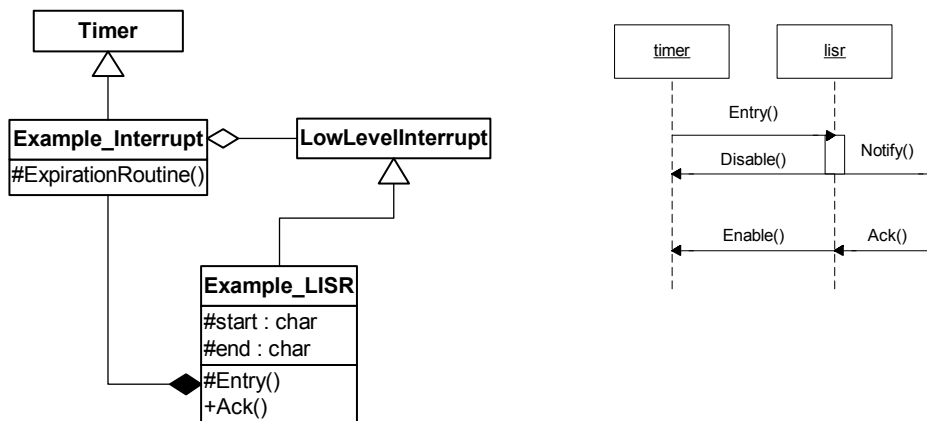


Inside the task's `Entry` member we are executing on a regular Nucleus PLUS task thread. Thus, all kernel services are available including self-suspension requests. This example task simply prints the buffer to the standard Nucleus C++ demonstration console and acknowledges the `lshr` object by executing its `Ack` member. Again thinking bare-bones C++, we are executing the `Example_LISR` classes `Ack` member on the regular task thread, not on any type of interrupt thread.

Interrupt Simulation

For demonstration purposes, we will simulate interrupts in order to allow this example to run on any supported Nucleus C++ PLUS target platform, including Nucleus C++ MNT.

The following UML diagram depicts our `Example_LISR` class in more detail. Notice our example composite contains a special timer that calls the `lshr` object's `Entry` member each time it expires, thus simulating the interrupt. In a real interrupt application, this will happen when the underlying interrupt vector fires.



Example Interrupt Simulation

The `Example_LISR::Entry` method reads the characters from a class scope array of text strings one-by-one until all of the strings in the array have been iterated. It then starts back at the beginning of the array and resumes this behavior. Below is a list of the text strings that we will present to the application. There are seven strings total, four are good and three are bad. The start character is `'.'` and the end character is `';`.



```

CHAR* Example_LISR::text[number_test_packets] =
{
    ":This is a good packet;",
    ":This is a bad packet that is missing end",
    ":This is a very nice packet;",
    "This is a bad packet that is missing start;",
    ":This is a very nifty packet;",
    "This is a very bad packet without start or end",
    ":This is the coolest packet ever;"
};

```

In our specific example, the timer will expire once every 100 milliseconds. The sequence diagram shown above indicates the timer will be disabled when the low-level interrupt activates the high-level interrupt. It will be re-enabled when the task acknowledges that it has serviced the buffer. Then, the timer starts again and more characters are read.

The following shows the output of the demonstration application. Notice the task only receives five of the seven buffers processed. What is happening?

Example Interrupt Application Output



The reason the task layer only receives five of the seven strings is due to the fact that the low-level interrupt processing spotted the two bad packets and they never made it to the high-level interrupt. The two bad packets were thrown out due to missing end characters. The high-level interrupt was never activated.

Of the five buffers the high-level interrupt received, it spotted the bad packet that our simple low-level interrupt processing did not. This packet was missing the start character that the low-level interrupt does not bother checking for. Good packets are sent to the ‘good’ task indicated by the “good-->” prompt and bad packets are sent to the ‘bad’ task indicated by the “bad-->” prompt. Prompts are built from the task’s name.

Interrupt Application Source Code Details

This section presents the example application source code and adds applicable comments to help the reader.

Simulated Interrupt Source

The `Example_Interrupt` class is shown below. This is a simple timer subclass that calls the `Entry` member of the supplied low-level interrupt object each time the timer expires. The `Timer` base class is parameterized to expire every 100 milliseconds and the timer is initially disabled. It will not fire until the base class `Enable` member is called.

```
class Example_Interrupt : public Timer
{
public:
    Example_Interrupt(LowLevelInterrupt* init_lisr)
    :    Timer("lisr_sim", TicksFromMs(100)/*initial*/,
           TicksFromMs(100)/*reschedule*/,
           FALSE/*initially disabled*/),
      lisr(init_lisr)
    { /*empty*/ }
    virtual ~Example_Interrupt() { /*empty*/ };

protected:
    // Data for each instance of timer.
    LowLevelInterrupt* lisr;

    // Expiration routine for the timer.
    virtual void ExpirationRoutine()
    {
        // Activate lisr interrupt.
        lisr->Entry();
    }
};
```



Although we present the `Example_LISR` class below, we will pause to show its `ReadChar` method since it is applicable to the simulation of packets. It simply scans the class scope text strings one-by-one and returns each character, one-by-one.

```
CHAR Example_LISR::ReadChar()
{
    // Move to the next buffer if necessary.
    if( *text_position == 0 )
    {
        if( ++text_number >= number_test_packets )
        {
            text_number = 0;
        }
        text_position = text[text_number];
    }

    // Return the next character and advance the pointer.
    return( *text_position++ );
}
```

LISR

The `Example_LISR` class declaration is shown below. This composite class contains the simulated interrupt source, the high-level interrupt object, and the buffer. This class also contains the simulation data and other state data that it uses to iterate the simulated character stream.

```
const INT lisr_buffer_size = 256;
const INT number_test_packets = 7;

class Example_LISR : public LowLevelInterrupt
{
public:
    // Constructor takes a vector and the start and end
    // characters that bound valid buffers.
    Example_LISR(INT vector, CHAR init_start_of_packet,
        CHAR init_end_of_packet)
        : LowLevelInterrupt(vector),
        buffer_position(buffer),
        start_of_packet(init_start_of_packet),
        end_of_packet(init_end_of_packet),
        hisr(this,init_start_of_packet,
            init_end_of_packet),
        text_number(0),text_position(text[0]),
        interrupt_source(this)
    {
        // Enable the timer after objects are created.
        interrupt_source.Enable();
    }
    virtual ~Example_LISR() { /*empty*/ }
    // virtual to acknowledge interrupt.
    virtual void Ack();
}
```



```

protected:
    // virtual to read a character.
    virtual CHAR ReadChar();
    // virtual Entry for LISR.
    virtual void Entry();
    // Data for each instance of LISR.
    Example_Interrupt    interrupt_source;
    Example_HISR         hisr;
    CHAR*                buffer_position;
    CHAR                 buffer[lisr_buffer_size];
    CHAR                 end_of_packet;
    CHAR                 start_of_packet;
    // Simulation data for each instance of LISR.
    static CHAR* text[number_test_packets];
    CHAR* text_position;
    INT text_number;
};

```

Notice the constructor takes the vector number as well as the packet boundary characters. Also notice that we are initializing data members in the member initialization list using the *this* pointer (both the `hisr` and the `interrupt_source` data members). Most compilers will warn about this since data member constructors might use the *this* pointer to operate on the object, and it does not fully exist at that point in time (it is being initialized). This is OK here since we know the classes only store away the pointer in their constructors. They do not operate on it.

For similar reasons, notice we do not enable the timer until all of the objects are completely created and initialized. We do not want the timer to start firing until the all of the objects in the picture completely exist. In a real application, this would be analogous to not allowing interrupts to fire until the handler object completely exists. This must be accounted for in your application.



Moving on to the definition of the class, below are the implementation details for the `Entry` member. Notice that we perform minimal processing and return to completion without any self-suspension requests. We also disable the simulation timer before `hissr` notification in order to prevent further interrupts from occurring until after the buffer has been serviced by the task.

```
void Example_LISR::Entry()
{
    // Read a char into the buffer.
    *buffer_position = ReadChar();
    // Check for end-of-packet.
    if( *buffer_position == end_of_packet )
    {
        // We are at the end, terminate it.
        *++buffer_position = NULL;
        // Stop the interrupt source.
        interrupt_source.Disable();
        // Setup buffer position for next time.
        buffer_position = buffer;
        // Notify hisr.
        hisr.Notify(buffer);
    }
    else
    {
        // Not the end, check for new start. Do not check
        // the first character in the buffer.
        if
        (
            (*buffer_position == start_of_packet)
            &&
            (buffer_position != buffer)
        )
        {
            // Oops! We missed a packet, start again.
            buffer_position = buffer;
            *buffer_position = start_of_packet;
        }
        // Advance pointer.
        buffer_position++;
    }
}
```

Below are the implementation details for the `Ack` member. All we need to do is re-enable the timer to start the interrupts flying again. This member is called by the task after it has serviced the last buffer.

```
void Example_LISR::Ack()
{
    // Start the interrupt source.
    interrupt_source.Enable();
}
```



HISR

The `Example_HISR` class declaration is shown below. This composite class contains the ‘good’ and the ‘bad’ tasks, the packet boundary characters, and a pointer to the buffer that it will eventually receive from the low-level interrupt object.

```
class Example_HISR : public HighLevelInterrupt
{
public:
    // Constructor takes end of packet character.
    Example_HISR(Example_LISR* interrupt_source,
        CHAR init_start_of_packet,
        CHAR init_end_of_packet)
    : HighLevelInterrupt("hisr"/*name*/,
        2/*priority*/,2048/*stacksize*/),
        good_task("good",interrupt_source),
        bad_task("bad",interrupt_source),
        start_of_packet(init_start_of_packet),
        end_of_packet(init_end_of_packet)
    {
        // Start our tasks.
        good_task.Start();
        bad_task.Start();
    }
    virtual ~Example_HISR() { /*empty*/ }

    // Called to notify HISR of new buffer.
    virtual void Notify(CHAR* new_buffer);

protected:
    // virtual Entry routine for this HISR.
    virtual void Entry();

    // Data for each instance of HISR.
    CHAR* buffer;
    CHAR start_of_packet;
    CHAR end_of_packet;
    Example_Task good_task;
    Example_Task bad_task;
};
```

Notice the constructor takes the packet boundary characters and a pointer to the low-level interrupt object that it is associated with. It uses this reference to create the tasks, passing the pointer to the task constructors via the member initialization list for the class.



Moving on to the definition of the class, below are the implementation details for the `Notify` member. Notice that we perform minimal processing and return to completion without any self-suspension requests. We simply capture the buffer pointer and perform a self-activation by calling the base class `Activate` operation (implicitly through the *this* pointer to operate on the particular instance of `Example_HISR`).

```
void Example_HISR::Notify (CHAR* new_buffer)
{
    buffer = new_buffer;
    Activate();
}
```

Below are the implementation details for the `Entry` member. This member is called automatically by the kernel on our behalf due to the activation request the low-level interrupt made in the `Notify` member shown above. The *this* pointer is automatically setup properly by the base class so all operations are performed on the local data specific to this instance of `Example_HISR` (for example, the `buffer` data member setup above in the `Notify` method).

Notice that we again perform minimal processing and return to completion without any self-suspension requests. The first part of the method finds the end of the buffer and the second checks for valid start and end characters. Depending on the results of the check, either the ‘good’ or the ‘bad’ task is notified with the current buffer.

```
void Example_HISR::Entry()
{
    // Find the end of the packet and back up to end character
    // position (-2 characters).
    CHAR* temp = buffer;
    while( *temp++ ) { /*empty*/ }
    temp -= 2;

    // Check if it is a good packet, indicated by the first
    // character being the start-of-packet and the last
    // character being the end-of-packet.
    if
    (
        (*temp == end_of_packet)
        &&
        (buffer[0] == start_of_packet)
    )
    {
        good_task.Notify(buffer);
    }
    else
    {
        bad_task.Notify(buffer);
    }
}
```



Task

The `Example_Task` class declaration is shown below. This class contains a synchronization event, a pointer to the buffer being serviced, and a pointer to the low-level interrupt object that is the source of the interrupt. This `l isr` pointer is stored so we can acknowledge the object after processing the buffer.

```
class Example_Task : public Task
{
public:
    // Constructor takes a name and a l isr.
    Example_Task(CHAR* name,
                Example_LISR* interrupt_source)
    : Task(name, 1536/*stacksize*/, 10/*priority*/,
          5/*timeslice*/, TRUE/*preempt*/),
      l isr(interrupt_source)
    { /*empty*/ }
    virtual ~Example_Task() { /*empty*/ };

    // Called to notify task of new buffer.
    virtual void Notify(CHAR* new_buffer);

protected:
    // Task entry member.
    virtual void Entry();

    // Data for each instance of task. We have an
    // event that indicates a new buffer, a
    // pointer to the buffer received, and a l isr
    // that will be the source of the interrupt.
    Event event;
    CHAR* buffer;
    Example_LISR* l isr;
};
```

Notice the constructor takes a name so we can distinguish the tasks on the printout and it also takes a pointer to the low-level interrupt object that the task is associated with.

Moving on to the definition of the class, below are the implementation details for the `Notify` member. Notice that we perform minimal processing and return to completion without any self-suspension requests. We simply capture the buffer pointer and set our instance's event by operating on it through its `Set` operation (implicitly through the *this* pointer to operate on the particular instance of `Example_Task`).

```
void Example_Task::Notify(CHAR* new_buffer)
{
    buffer = new_buffer;
    event.Set();
}
```



Below are the implementation details for the `Entry` member. The *this* pointer is automatically setup properly by the base class so all operations are performed on the local data specific to this instance of `Example_Task` (for example, the buffer data member setup above in the `Notify` method).

The simple task behavior is to continuously loop, suspending on our local event object until it has been set by the `Notify` operation described above. The default parameters of the `Retrieve` method of class `Event` specify event consumption and indefinite suspension until the event has been set.

```
void Example_Task::Entry()
{
    // Setup console and initialize our info.
    UpdateInfo();
    ostream_Npp& cout_Npp =
        *NppStandardStreamsComponent()->cout();

    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Wait for notification.
        event.Retrieve();

        // Service the buffer.
        cout_Npp << endl << info.GetName()
            << "--> " << buffer << flush;

        // Ack the interrupt source.
        lisr->Ack();
    }
}
```

Once the event is set and the task is marked as *ready*, the kernel resumes normal task scheduling and the task will run at its priority within the rest of the application. The task prints its name prompt and the buffer to the standard Nucleus C++ demonstration console. It then acknowledges the interrupt source so another buffer can be captured.



Creating the Objects

To complete the picture of the interrupt example, we need to create the objects and start them along their merry way. This is shown below.

```
void
NppCreate( void* /* first_available_memory */ )
{
    // Empty.
}

void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS();
    nppPLUS->Initialize();

    // Nucleus C++ standard STREAMS component.
    CreateNppSTDSTREAMS()->Initialize();

    #if (NU_STATIC_OBJECT_SUPPORT)
        // Nucleus C++ STATIC component.
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    // Create an interrupt object.
    new Example_LISR( 7/*vector*/, ':'/*start*/, ';'/*end*/ );
}
```

Notice all we need to do is create the interrupt object since the classes are designed as composites that contain the dependent objects. The steps required to start the tasks and other necessary details are handled by the designer of the class, not the user of the class.



Accessing HighLevelInterrupt Information

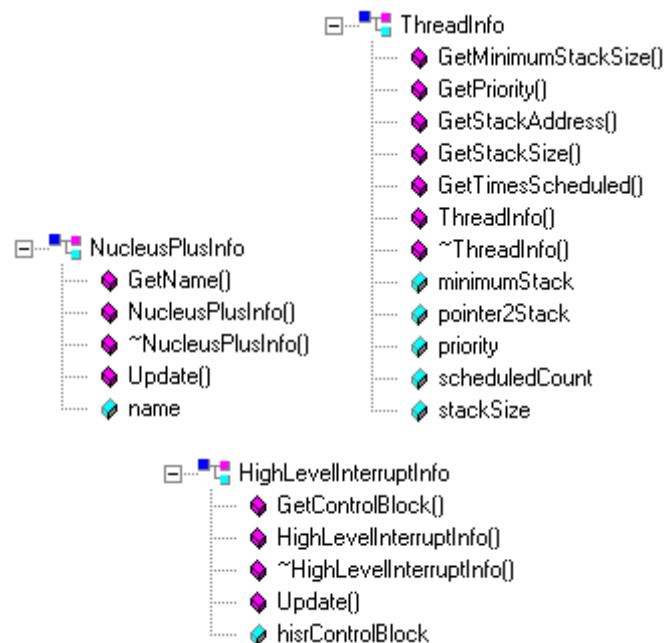
Each class `HighLevelInterrupt` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `HighLevelInterruptInfo` object that has been filled with updated information about the specific interrupt instance.

```
// Get a reference to an updated info object from the interrupt.
const HighLevelInterruptInfo& info =
    myHighLevelInterrupt.UpdateInfo();

// Do some things with the information.
UNSIGNED minimum_stack_size = info.GetMinimumStackSize();
UNSIGNED priority = info.GetPriority();
UNSIGNED stack_size = info.GetStackSize();
UNSIGNED times_scheduled = info.GetTimesScheduled();
```

Once the interrupt info object is retrieved, you can simply invoke the members of the `HighLevelInterruptInfo` object to retrieve the following information: the control block; the minimum size the stack has ever reached; the priority of the interrupt; the stack size; the number of times the interrupt has been scheduled; etc.

The following shows the structure of the class `HighLevelInterruptInfo`, a subclass of `ThreadInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class HighLevelInterruptInfo

Please see the *Class Reference* chapter for details on how to use specific members.



Software Interrupts

The Nucleus C++ PLUS class `signal` is a class that manages asynchronous software interrupts in an embedded application using the Nucleus PLUS signaling services. This is a synchronization mechanism used for inter-task communication. The class `Task` has special software to support this advanced mechanism when enabled.

Class Signal Overview

Signals are in some ways similar to event flags. However, there are significant differences in operation. `Event flag` usage is synchronous by nature. The task does not recognize the event flags are present until the specific service request is made. Signals are operated in an asynchronous manner. When a signal is present, the task is interrupted and a special signal handling routine (a member function of class `Signal`) is executed. Each task is capable of handling 30 signals.

Signal Handling Routine

Processing inside the signal-handling member has virtually the same constraints as a high-level interrupt service routine. Basically, most Nucleus C++ services are available, provided self-suspension is avoided.

Enable Signal Handling

By default, tasks are created with all signals disabled. Individual signals may be enabled and disabled dynamically by each task.

Clearing Signals

Signals are automatically cleared when signal handling is invoked. Additionally, signals are cleared when a solicited request to receive signals is made. **NOTE:** tasks cannot suspend on solicited requests to receive signals.

Multiple Signals

Signals for a task are cleared once the signal-handling routine is started. Signal handling routines are not interrupted by new signals. Processing of any new signals takes place after the current signal-processing completes. Identical signals sent before the first signal is recognized are discarded.

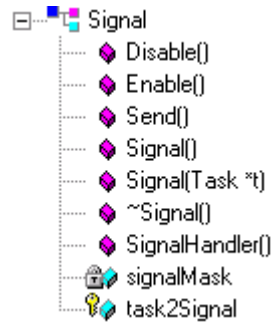


Determinism

Processing time required to send and receive signals is constant, at least in the worst case. Of course, the time required to execute a signal-handling member is specific to the derived signal and the application needs.

Structure of Class Signal

The following diagram shows the structure of class `Signal`.



Structure of class Signal

IMPORTANT NOTE: In order to run the example presented in this section, and to use signals in your application, you must turn on signal support for class `Task`. This is accomplished by setting the flag `NU_TASK_SIGNAL_SUPPORT` in your `NPPAPP.H` file to a 1. Please see the “Optimizations for Embedded Systems” section in this manual for more information on this and other flags.

Using Software Interrupts, 1-2-3

Using signals in Nucleus C++ is very easy. The use is very similar to the way other *thread* classes are created and used. Simply derive from the Nucleus C++ PLUS class `Signal` and provide an implementation of the virtual `SignalHandler` member.

Class `Signal` is an abstraction of the mechanisms required to receive a signal handler callback from the underlying RTOS. You can operate on your objects using the members of class `Signal`.



Subclass class Signal

The first step of specializing the signal is to subclass the base class `Signal`. The following example specifies a signal that takes a name as a constructor parameter. This name is stored away in a data element for subsequent use within the signal handler.

```
class Example_Signal : public Signal
{
public:
    // Using this version of the constructor, the signal
    // is created for the current executing task.
    Example_Signal (CHAR* init_name)
    :    name (init_name)
    { /*empty*/ }

    // Alternatively, using this version of the
    // constructor, the signal is created for task 'task'.
    Example_Signal (CHAR* init_name, Task *task)
    :    Signal (task), name (init_name)
    { /*empty*/ }

protected:
    // Signal handler is executed for each signal sent.
    virtual void SignalHandler();

    // Data for each instance of Example_Signal.
    CHAR* name;
};
```

There are two constructors, the first creates the signal for the calling task and the second creates the signal explicitly for the task specified. In the first case, there is nothing to do since the base class default constructor creates the signal for the calling task.

The second version of the constructor passes the task pointer to the base class in the base class initializer. This version of the base class constructor sets up the appropriate logic to associate the signal with the specified task.



When we speak about signal and task association, what we are saying is that the signal handler will execute on top of the task context that it is associated with. The signal handler routine will be called using the task's stack and it will execute in lieu of the task, when that task is scheduled to run from the perspective of the RTOS (at the priority of the task, when the task is ready, and in place of the task). This is essentially an asynchronous software interrupt for the task thread, not the system as a whole.

The base class `Signal` creates a Nucleus PLUS signal service in its constructor and removes the real-time signal service in its destructor. The data and operations required to manage the Nucleus PLUS signal service are handled automatically. The derived class simply adds data and behavior that is specific to the particular signal extension.

Define Signal Handler Behavior

The `SignalHandler` member is the callback where the derived class receives the signal thread of execution from Nucleus PLUS. The following example simply prints the name of the signal to the standard Nucleus C++ demonstration console for identification. `SignalHandler` is called by the base class with the *this* pointer properly setup to point to the specific signal instance (and therefore its copy of instance data).

```
void
Example_Signal::SignalHandler()
{
    // Print the name of the signal.
    ostream_Npp& cout_Npp =
        *NppStandardStreamsComponent()->cout();
    cout_Npp << name << flush;
}
```

Create a Simple Quarterback and Receiver of Signals

Now that we have a signal class, let's declare a few tasks that demonstrate the capability. The signals will be associated with the receiver task since the signal will be *sent* to the receiver by the quarterback. We will introduce a third task in the system that sits and spins in order to demonstrate how signals work. For ease of reading, we will define three priority constants:

```
const OPTION low_priority = 255;
const OPTION medium_priority = 25;
const OPTION high_priority = 10;
```



Quarterback

The quarterback class declaration is shown below. It is a simple Task subclass that takes a receiver task as a constructor parameter and stores it away for use in the Entry method. Notice the task is created at the highest priority.

```
class Signal_Quarterback : public Task
{
public:
    Signal_Quarterback(Signal_Receiver* init_receiver)
    :    Task("QB"/*name*/,1536/*stacksize*/,
          high_priority/*priority*/,5/*timeslice*/,
          TRUE/*preemption*/ ),
      receiver(init_receiver)
    { /*empty*/ }
    virtual ~Signal_Quarterback() { /*empty*/ }

protected:
    // Entry member.
    virtual void Entry();

    // Data for each instance of Signal_Quarterback.
    Signal_Receiver* receiver;
};
```

The Entry method for the quarterback class simply sleeps for 100 milliseconds then sends the receiver both of its signals. These signals are described in more detail below when we describe the receiver class.

```
void
Signal_Quarterback::Entry()
{
    BOOL bKeepLooping = TRUE;
    while( bKeepLooping )
    {
        // Sleep for 100 milliseconds.
        Sleep(TicksFromMs(100));

        // Send the receiver the signals.
        receiver->SendSignal_A();
        receiver->SendSignal_B();
    }
}
```



Simple Spinner Task

In order to fully demonstrate signal and task association, we will introduce a third task type into the application, the `Spinner` class. This is a simple `Task` subclass that continuously spins in the `Entry` method without yielding (relinquishing) or suspending. The class is shown below. Notice it is created at medium priority.

```
class Spinner_Task : public Task
{
public:
    Spinner_Task()
    : Task("spin"/*name*/,1536/*stacksize*/,
          medium_priority/*priority*/,
          5/*timeslice*/,TRUE/*preemption*/)
    { /*empty*/ }
    virtual ~Spinner_Task() { /*empty*/ }

protected:
    // Entry member.
    virtual void Entry()
    {
        // Loop continuously in the background.
        BOOL bKeepLooping = TRUE;
        while( bKeepLooping )
        {
            // Spin at the medium priority.
        }
    }
};
```



Receiver

The receiver class declaration is shown below. The task will behave just like the spinner task (`Signal_Receiver` is a subclass of `Spinner_Task`) except that it will add a quarterback data element and two signals. There are two signals in order to demonstrate both ways to create a signal for association with a particular task.

Since it is a direct derivation of the spinner task, it inherits the initial medium priority that the spinner class specifies. This will be changed in the `Entry` method to further examine signal characteristics.

```
class Signal_Receiver : public Spinner_Task
{
    public:
        Signal_Receiver()
        :   signal_A("A",this),quarterback(this)
        { /*empty*/ }
        virtual ~Signal_Receiver() {delete signal_B;}

        // Methods to send the two signals.
        void SendSignal_A();
        void SendSignal_B();

    protected:

        // Data for each instance of task.
        Example_Signal signal_A;
        Example_Signal* signal_B;
        Signal_Quarterback quarterback;

        // Entry member.
        virtual void Entry();
};
```

Notice `signal_A` is an object and `signal_B` is just a pointer. Thus, we create and initialize `signal_A` within the member initialization list of the constructor (we will create and initialize `signal_B` in the `Entry` method and will destroy it in the destructor). What this means is that `signal_A` will be created on the thread that creates the receiver task. However, we associate it with the receiver task by passing the *this* pointer into the member initializer.

The quarterback is also initialized by passing the *this* pointer of the receiver into the member initializer for the quarterback task. This is how the quarterback knows which receiver to signal within its `Entry` method.



The methods for the receiver class are shown below. The signal sending methods simply send the signal to the appropriate signal data element. The `Entry` method for the receiver creates the second signal, enables both signals, starts the quarterback, and then calls the base class `Entry` method to perform the base sit and spin behavior.

```
void
Signal_Receiver::SendSignal_A()
{
    signal_A.Send();
}

void
Signal_Receiver::SendSignal_B()
{
    signal_B->Send();
}

void
Signal_Receiver::Entry()
{
    // Create signal B for the current executing task (this).
    signal_B = new Example_Signal("B");

    // Enable the signals.
    signal_A.Enable();
    signal_B->Enable();

    // Start the quarterback now that our signals exist.
    quarterback.Start();

    // Spin at the lowest priority.
    //COMMENTED OUT!!! ChangePriority(low_priority);

    // Simply call into the base class entry to spin.
    Spinner_Task::Entry();
}
```

Notice `signal_B` is created and initialized using the default constructor that associates the signal being created with the thread that the constructor is called on. Since we are in *this* task's `Entry` method, we are executing on its task thread. Thus, `signal_B` is automatically associated with the receiver task.

Notice there is a commented out call that changes the priority of the receiver task to the lowest priority in the system before the task spins. This is marked with “COMMENTED OUT!!!” in order for the application to behave like we want it to.



Create Objects

The final step is to create the tasks and start them. The section of code below shows we are only creating two tasks, the spinner and the receiver. The third task is automatically created within the receiver task. Thus, we have three tasks executing, one at a high priority (the quarterback) and the other two at medium priority (the spinner and receiver).

```
void
NppCreate( void* /*first_available_memory*/ )
{
    // Empty, initialize on a task thread.
}

void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS();
    nppPLUS->Initialize();

    // Create a Nucleus C++ standard STREAMS object.
    CreateNppSTDSTREAMS()->Initialize();
    ostream_Npp* cout_Npp =
        NppStandardStreamsComponent()->cout();

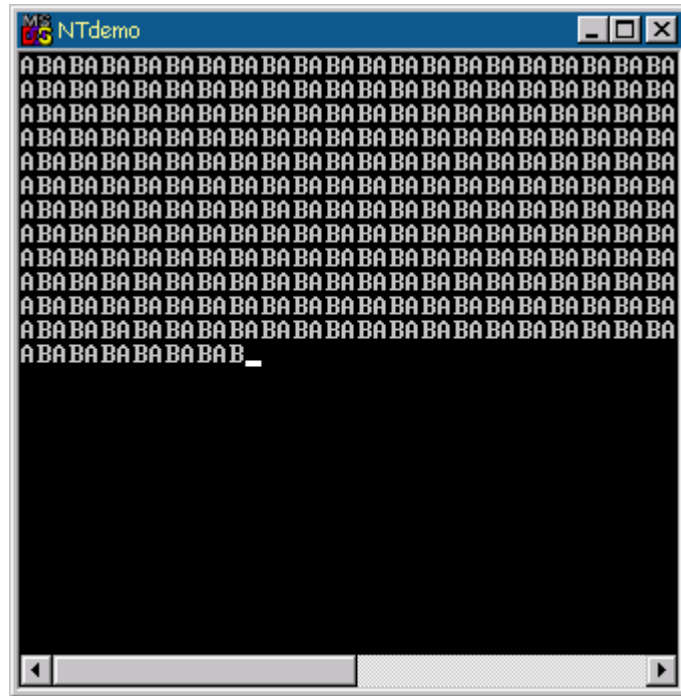
    // Initialize static objects.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    // Create and start the tasks.
    Spinner_Task* spinner = new Spinner_Task;
    spinner->Start();

    Signal_Receiver* receiver = new Signal_Receiver;
    receiver->Start();
}
```



The output of this demonstration application is shown below. Notice we get a nice continuous stream of handled signals, evidenced by the fact that the only printout in the application is within the signal handlers. What is not clear by the output is *when* are these signal handlers being executed and *where* are they running?



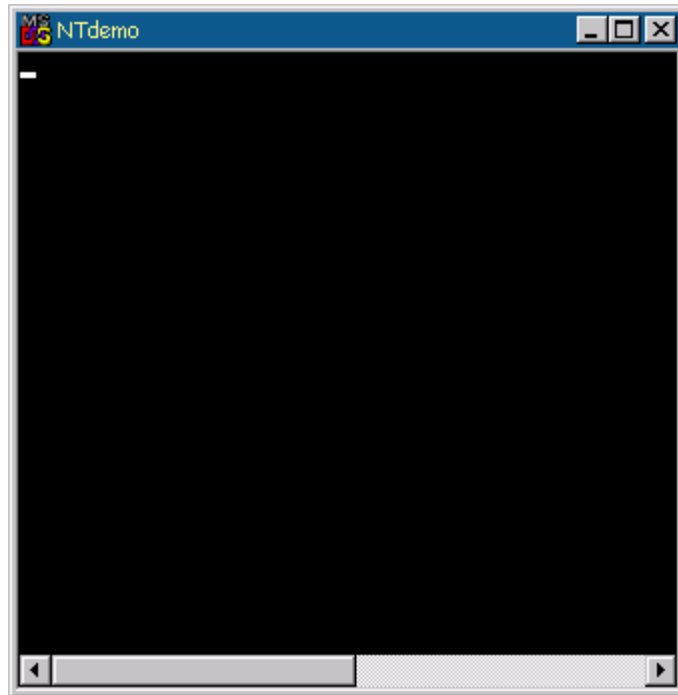
Signal Application Output

The answer to the *when* question is the signal will execute when the receiver task is scheduled to run within the Nucleus PLUS scheduler. As mentioned above, this is because the signal has been *associated* with the receiver. The signal handlers fire because the quarterback sent the signals (they are *present* when the receiver is ready to run).

Where they execute is more of a question of what stack do they run on. They run on the receiver's stack under the receiver task's context.



To fully understand signal handlers will only run when the task they are associated with is ready to run, uncomment the commented out line above and observe the application output shown below.



Signal Application Output with Receiver at Lowest Priority

What is happening? The reason no signal handler routines are executing in this version of the application (with the receiver changed to low priority) is because the receiver task is never scheduled to run from the perspective of the Nucleus PLUS kernel. Why? Because the medium priority spinner task is always running, starving the receiver task from being able to get a time-slice.

When the receiver is kept at the same medium priority as the spinner task, the time-slicing feature of Nucleus PLUS periodically schedules both the receiver and the spinner. They are effectively executing concurrently. When a signal is present *and* the receiver receives its time-slice, the signal handler fires and prints the messages to the console.



Special Memory Management Classes

For more information on the standard C++ memory facilities provided by Nucleus C++ BASE, please refer to the user's guide section of the Nucleus C++ BASE Reference manual for more information.

Class `MemoryPool` Overview

A dynamic memory pool object contains a user-specified number of bytes. The memory location of the pool is optionally determined by the application. If the memory location of the pool is not provided, a memory block of the appropriate size is allocated from the free store. Variable-length allocation and deallocation services are provided for the dynamic memory pool. Allocations are performed in a first-fit manner, i.e. the first available memory that satisfies the request is allocated. If the allocated block is significantly larger than the request, the unused memory is returned to the dynamic memory pool. Previously deallocated blocks are merged during allocation searching.

Suspension

The `Allocate` member provides options for unconditional suspension, suspension with a time-out, and no suspension. A task attempting to allocate dynamic memory from a pool that does not currently have enough available memory may suspend. Resumption of the task is possible when enough previously allocated memory is returned to the pool. Multiple tasks may suspend on a single dynamic memory pool. Tasks are suspended in either FIFO or priority order, depending on how the dynamic memory pool was created. If the dynamic memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the dynamic memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Dynamic Creation

Nucleus PLUS dynamic memory pool objects are created and deleted dynamically. There is no preset limit on the number of `MemoryPool` objects an application may have.



Determinism

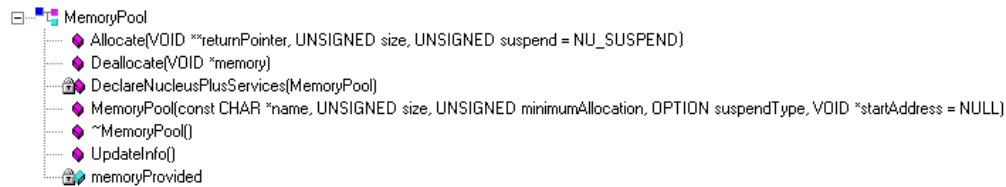
Allocating memory from a dynamic memory pool is inherently indeterministic. This is largely due to the possible memory fragmentation within the pool. The first-fit algorithm is basically a linear search, and as a result the worst-case performance depends on the amount of fragmentation. However, memory deallocation is constant. Processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the dynamic memory pool.

Information

Application tasks may obtain a list of active dynamic memory pools. Detailed information about each dynamic memory pool is also available. This information includes the dynamic memory pool name, starting pool address, total size, free bytes, number of tasks suspended, and the identity of the first suspended task.

Structure of Class `MemoryPool`

The following diagram shows the structure of class `MemoryPool`.



Structure of class `MemoryPool`

Using Memory Pools, 1-2-3

Using memory pools in Nucleus C++ PLUS is easy. You can simply use the class as is, there is no need to subclass (unless you want to extend the class). The advantage partition pools have over memory pools is they do not fragment. The advantage memory pools have over partition pools is they satisfy arbitrary sized memory requests.



Create the Memory Pool

The following example creates a memory pool object that is a megabyte large and has a minimum allocation size of a kilobyte. This means that if you request a byte, it will mark a kilobyte as *allocated* and pass you a pointer to it. This example allocates the memory for the pool automatically from the C++ free store. The suspension parameter is similar to all other Nucleus PLUS suspension parameters and specifies how tasks suspend on the service if allocation requests cannot be immediately satisfied.

```
MemoryPool pool
(
    "pool",          // name.
    1000*1024,       // memory pool size.
    1024,            // minimum allocation size.
    NU_PRIORITY      // suspension type.
    NULL             // allocate pool memory from the C++ free store.
);
```

As another example, the following code snippet creates a similar memory pool, except the memory that the pool manages (works out of) is explicitly provided as a constructor parameter.

```
MemoryPool pool
(
    "pool",          // name.
    1000*1024,       // memory pool size.
    1024,            // minimum allocation size.
    NU_PRIORITY      // suspension type.
    0x1FFF0000       // pointer to pool memory.
);
```

Allocating Memory

Once a memory pool exists, allocating memory from it is easy. The following code snippets demonstrate several methods of allocation. All of the allocations request a 256-byte block from the pool. The status return value can be inspected to determine if the allocation succeeded. Please refer to the reference chapter on the `Allocate` member for more information of all possible return values and their meanings.

```
STATUS status;
VOID* memory;

// This call conditionally suspends if the allocation request
// cannot be immediately satisfied.
status = pool.Allocate( &memory, 256, NU_SUSPEND );

// This call does not suspend at all if the allocation request
// cannot be immediately satisfied.
status = pool.Allocate( &memory, 256, NU_NO_SUSPEND );

// This call conditionally suspends for 100 milliseconds or
// until the allocation can be satisfied, whichever is first.
status = pool.Allocate( &memory, 256, TicksFromMs( 100 ) );
```



Returning Memory

Once memory has been successfully allocated from a memory pool and it is finished being used, it can be returned to the pool using the `DeAllocate` operation of class `MemoryPool`. The following code snippet demonstrates how to return the memory allocated in the above example.

```
// Return the memory to the pool.  
pool.Deallocate( memory );
```

Accessing MemoryPool Information

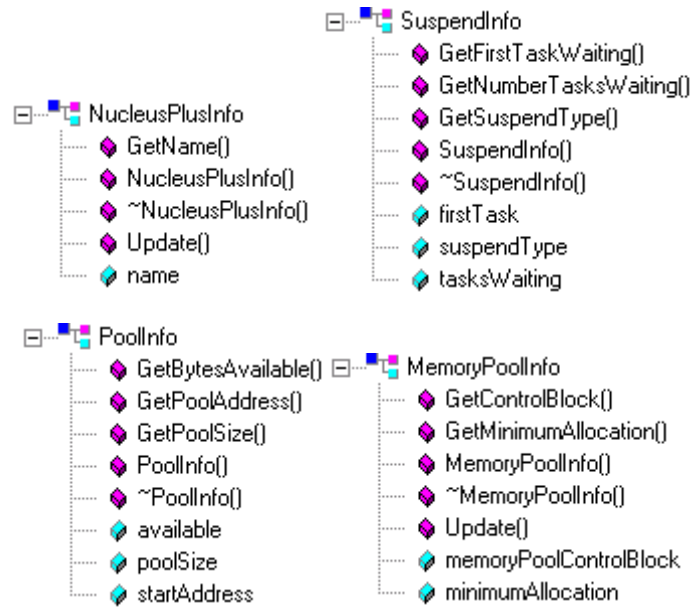
Each class `MemoryPool` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `MemoryPoolInfo` object that has been filled with updated information about the specific memory pool instance.

```
// Get a reference to an updated info object from the memory pool.  
const MemoryPoolInfo& info = myMemoryPool.UpdateInfo();  
  
// Do some things with the information.  
UNSIGNED minimum_allocation = info.GetMinimumAllocation();  
UNSIGNED bytes_available = info.GetBytesAvailable();  
UNSIGNED pool_size = info.GetPoolSize();  
UNSIGNED number_tasks_waiting = info.GetNumberTasksWaiting();
```

Once the memory pool info object is retrieved, you can simply invoke the members of the `MemoryPoolInfo` object to retrieve the following information: the control block; the minimum number of bytes for each allocation from this pool; how big the pool is; how many bytes are available; how many tasks are suspended on the pool; etc.



The following shows the structure of the class `MemoryPoolInfo`, a subclass of `PoolInfo`, a subclass of `SuspendInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class `MemoryPoolInfo`

Please see the *Class Reference* section for details on how to use specific members.

Class PartitionPool Overview

A partition memory pool object contains a specific number of fixed-size memory partitions. The memory location of the pool, the number of bytes in the pool, and the number of bytes in each partition are determined by the application. If the memory location of the pool is not provided, a memory block of the appropriate size is allocated from the free store. Individual partitions are allocated and deallocated from the partition-memory pool.



Suspension

The `Allocate` member provides options for unconditional suspension, suspension with a time-out, and no suspension. A task attempting to allocate a partition from an empty pool can suspend. Resumption of that task is possible when a partition is returned to the pool.

Multiple tasks may suspend on a single partition memory pool. Tasks are suspended in either FIFO or priority order, depending on how the partition memory pool was created. If the partition memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the partition memory pool supports priority suspension, tasks are resumed from high priority to low priority. All Nucleus C++ members that provide suspension options have their suspend parameter set to `NU_SUSPEND` by default. This means that if the parameter is absent (`orNU_SUSPEND`), tasks are suspended if the request cannot be satisfied.

Dynamic Creation

Nucleus C++ partition memory pool objects are created and deleted dynamically. There is no preset limit on the number of `PartitionPool` objects an application may have.

Determinism

Since searching is completely avoided, processing required for allocating and deallocating partitions is fast and constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the partition memory pool.

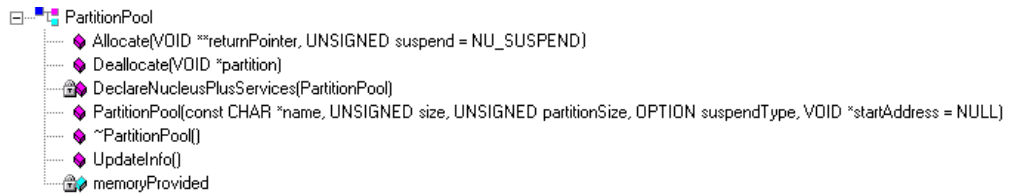
Partition Information

Application tasks may obtain a list of active partition memory pools. Detailed information about each partition memory pool is also available. This information includes the partition memory pool name, starting pool address, total partitions, partition size, remaining partitions, number of tasks suspended, and the identity of the first suspended tasks.



Structure of Class PartitionPool

The following diagram shows the structure of class PartitionPool.



Structure of class PartitionPool

Using Partition Pools, 1-2-3

Using partition pools in Nucleus C++ PLUS is easy. You can simply use the class as is, there is no need to subclass (unless you want to extend the class). Again, the advantage partition pools have over memory pools is they do not fragment. The advantage memory pools have over partition pools is they satisfy arbitrary sized memory requests.

Create the Partition Pool

The following example creates a partition pool object that is two kilobytes large and has a partition size of fifty bytes. This means that every request will return a block that is fifty bytes long if successful. This example allocates the memory for the pool automatically from the C++ free store. The suspension parameter is similar to all other Nucleus PLUS suspension parameters and specifies how tasks suspend on the service if allocation requests cannot be immediately satisfied.

```

PartitionPool pool
(
    "pool", // name.
    2*1024, // total pool size.
    50,     // partition size.
    NU_PRIORITY, // suspension ordering
    NULL    // allocate pool memory from the C++ free store.
);
  
```

As another example, the following code snippet creates a similar partition pool, except the memory that the pool manages (works out of) is explicitly provided as a constructor parameter.

```

PartitionPool pool
(
    "pool", // name.
    2*1024, // total pool size.
    50,     // partition size.
    NU_PRIORITY, // suspension ordering
    0xFFFF0000 // allocate pool memory from the C++ free store.
);
  
```



Allocating Memory

Once a partition pool exists, allocating memory from it is easy. The following code snippets demonstrate several methods of allocation. The status return value can be inspected to determine if the allocation succeeded. Please refer to the reference chapter on the `Allocate` member for more information of all possible return values and their meanings.

```
STATUS status;
VOID* memory;

// This call conditionally suspends if the allocation request
// cannot be immediately satisfied.
status = pool.Allocate( &memory, NU_SUSPEND );

// This call does not suspend at all if the allocation request
// cannot be immediately satisfied.
status = pool.Allocate( &memory, NU_NO_SUSPEND );

// This call conditionally suspends for 100 milliseconds or
// until the allocation can be satisfied, whichever is first.
status = pool.Allocate( &memory, TicksFromMs( 100 ) );
```

Returning Memory

Once memory has been successfully allocated from a partition pool and it is finished being used, it can be returned to the pool using the `DeAllocate` operation of class `PartitionPool`. The following code snippet demonstrates how to return the memory allocated in the above example.

```
// Return the memory to the pool.
pool.Deallocate( memory );
```

Accessing PartitionPool Information

Each class `PartitionPool` instance has an information object that allows you to obtain information about the object at a given snapshot in time. The `UpdateInfo` member is used to retrieve a reference to a class `PartitionPoolInfo` object that has been filled with updated information about the specific partition pool instance.

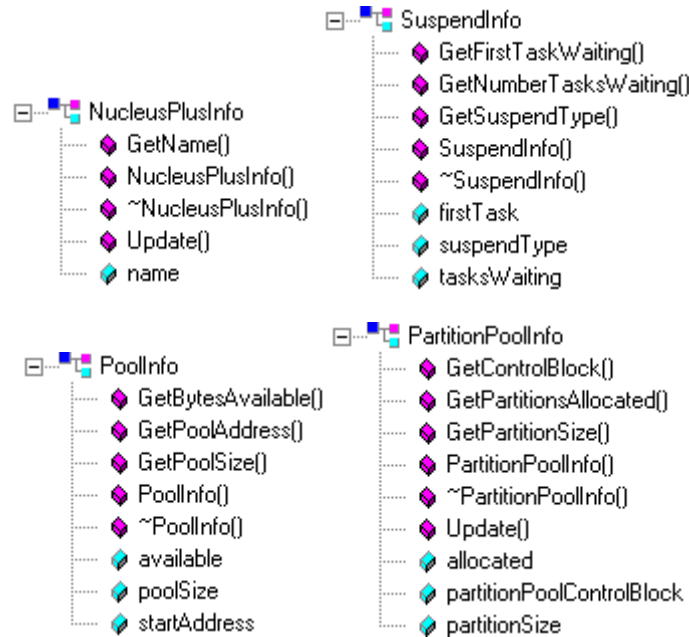
```
// Get a reference to an updated info object from the pool.
const PartitionPoolInfo& info = myPartitionPool.UpdateInfo();

// Do some things with the information.
UNSIGNED partition_size = info.GetPartitionSize();
UNSIGNED partitions_allocated = info.GetPartitionsAllocated();
UNSIGNED bytes_available = info.GetBytesAvailable();
UNSIGNED pool_size = info.GetPoolSize();
UNSIGNED number_tasks_waiting = info.GetNumberTasksWaiting();
```



Once the `PartitionPool Info` object is retrieved, you can simply invoke the members of the `PartitionPoolInfo` object to retrieve the following information: the control block; the number of bytes for each partition in this pool; how many allocations are in the pool; how big the pool is; how many bytes are available; how many tasks are suspended on the pool; etc.

The following shows the structure of the class `PartitionPoolInfo`, a subclass of `PoolInfo`, a subclass of `SuspendInfo`, which is a subclass of `NucleusPlusInfo`:



Structure of class `PartitionPoolInfo`

Please see the Class Reference chapter for details on how to use specific members.





2

Class Reference



class Box : public NucleusPlus

Mailboxes provide a low-overhead mechanism to transmit simple messages. Each mailbox is capable of holding a single message the size of four 32-bit words. Messages are sent and received by value. A send message request copies the message into the mailbox, while a receive message request copies the message out of the mailbox.

Suspension

Send and receive mailbox services provide options for unconditional suspension, suspension with a time-out, and no suspension. Task objects can suspend on a mailbox for several reasons. A task attempting to receive a message from an empty mailbox can suspend. Also, a task attempting to send a message to a non-empty mailbox can suspend. A suspended task is resumed when the mailbox is able to satisfy that task's request. For example, suppose a task is suspended on a mailbox waiting to receive a message. When a message is sent to the mailbox, the suspended task is resumed. Multiple tasks can suspend on a single mailbox. Tasks are suspended in either FIFO or priority order, depending on how the mailbox was created. If the mailbox supports FIFO suspension, Tasks are resumed in the order in which they were suspended. Otherwise, if the mailbox supports priority suspension, tasks are resumed from high priority to low priority. If a Nucleus C++ Box member has a suspension option, this option is set to `NU_SUSPEND` by default. This means that if the parameter is absent, tasks are suspended if the request cannot be satisfied.

Broadcast

A mailbox message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the mailbox are given the broadcast message.

Dynamic Creation

Nucleus C++ mailboxes are created and deleted dynamically. There is no preset limit on the number of Box objects an application may have.

Determinism

Processing time required for sending and receiving mailbox messages is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the mailbox.



Mailbox Information

Application tasks may obtain a list of active mailboxes. Detailed information about each mailbox can also be obtained. This information included the mailbox name, suspension type, whether a message is present, and the first task waiting.

Public Member Functions

Member	Overview
<code>~Box</code>	Destructor
<code>Box</code>	Constructor
<code>Broadcast</code>	Broadcasts a message to all tasks waiting for a message from this <code>Box</code> object.
<code>Receive</code>	Retrieves a message from this <code>Box</code> object.
<code>Reset</code>	Discards message currently in this <code>Box</code> object.
<code>Send</code>	Places a message in this <code>Box</code> object.
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the <code>info</code> object.



Box::~Box

virtual

```
Box::~~Box();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete box service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Box : Box

Box : Box

```
(
const CHAR* name,
    OPTION suspend
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Constructs a task communication Box object. This requires creating a mailbox service in the kernel. The object is added to the classes list of instances.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The box's name.
suspend	Specifies how tasks suspend on the mailbox.

Return Value

None

Example

```
// create a new box which tasks suspend on in fifo
// order
Box* pBox = new Box("mybox", NU_FIFO );

// create a new box which tasks suspend on in
// priority order
Box* pBox = new Box("mybox",NU_PRIORITY );
```



Box::Broadcast

```

inline
STATUS
Box::Broadcast
(
    VOID* object,
    UNSIGNED suspend = NU_SUSPEND
);

```

Broadcasts a message to all tasks waiting for a message from this `Box` object. If the mailbox is empty, the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox broadcast service
Post-condition	If successful, the message was broadcast

Parameters

Parameter	Overview
object	A pointer to the message to broadcast.
suspend	Specifies whether or not to suspend the calling task if the mailbox is full. <code>NU_NO_SUSPEND</code> causes the service to return immediately regardless of whether the request can be satisfied. <code>NU_SUSPEND</code> causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.

Return Value

Return Value	Overview
<code>NU_INVALID_ENABLE</code>	Indicates the enable parameter is invalid.
<code>NU_INVALID_MAILBOX</code>	Indicates the mailbox pointer is invalid.
<code>NU_INVALID_POINTER</code>	Indicates the message pointer is NULL.
<code>NU_INVALID_SUSPEND</code>	Indicates that suspend attempted from a non-task thread.
<code>NU_MAILBOX_DELETED</code>	Mailbox was deleted while the task was suspended.
<code>NU_MAILBOX_FULL</code>	Indicates the mailbox is full.
<code>NU_MAILBOX_RESET</code>	Mailbox was reset while the task was suspended.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_TIMEOUT</code>	Indicates that the mailbox is still unable to accept the message even after suspending for the specified timeout value.



Example

```
UNSIGNED  message[4];

Box* pBox = new Box("mybox",NU_PRIORITY);

// assume that there are a number of tasks
// suspended on the box waiting for a message
// and you want to send them all a message at
// once.  Suspend for up to 10 ticks trying
// to broadcast the message.

if ( pBox->Broadcast(&message, 10) == NU_SUCCESS )
{
    // we successfully delivered the message
}
```



Box::Receive

```
inline
STATUS
Box::Receive
(
    VOID* destination,
    UNSIGNED suspend = NU_SUSPEND
);
```

Retrieves a message from this `Box` object. If the mailbox contains a message, the message is immediately removed from the mailbox and copied into the destination location.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox receive service.
Post-condition	If successful, the message is received and removed from the box.

Parameters

Parameter	Overview
destination	Points to the message destination. Note: the message destination must be at least the size of four <code>UNSIGNED</code> data elements. <code>suspend</code> specifies how tasks suspend on the service.
suspend	Specifies whether or not to suspend the calling task if the mailbox is full. <code>NU_NO_SUSPEND</code> causes the service to return immediately regardless of whether the request can be satisfied. <code>NU_SUSPEND</code> causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_MAILBOX_DELETED	Mailbox was deleted while the task was suspended.
NU_MAILBOX_EMPTY	Indicates the mailbox is empty.
NU_MAILBOX_RESET	Mailbox was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the mailbox is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// Box* pBox was created elsewhere...

UNSIGNED message[4];
STATUS status;

// wait for a message for up to 1500 milliseconds.
// Use the TicksFromMS macro to translate between
// time and ticks, which is what receive expects.

status = pBox->Receive(&message, TicksFromMS(1500));

if (status == NU_SUCCESS)
{
    // we received a message, handle it
}
else
if (status == NU_TIMEOUT)
{
```



Box::Reset

```
inline  
STATUS  
Box::Reset();
```

Discards message currently in this `Box` object. All tasks suspended on the mailbox are resumed with the appropriate reset status.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox reset service.
Post-condition	If successful, the mailbox is empty.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_MAILBOX</code>	Indicates the mailbox pointer is invalid.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// Box mailbox created elsewhere  
  
// this call will resume any task suspended on the  
// given mailbox with the appropriate error code  
// and discard any data in the mailbox.  
  
mailbox.Reset();  
    // you timed out waiting for the message  
}
```



Box::Send

```

inline
STATUS
Box::Send
(
    VOID* object,
    UNSIGNED suspend = NU_SUSPEND
);

```

Places a message in this `Box` object. If the mailbox is empty the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox send service.
Post-condition	If successful, the message was sent to the mailbox.

Parameters

Parameter	Overview
object	A pointer to the message to send.
suspend	Specifies whether or not to suspend the calling task if the mailbox is full. <code>NU_NO_SUSPEND</code> causes the service to return immediately regardless of whether the request can be satisfied. <code>NU_SUSPEND</code> causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_MAILBOX_DELETED	Mailbox was deleted while the task was suspended.
NU_MAILBOX_FULL	Indicates the mailbox is full.
NU_MAILBOX_RESET	Mailbox was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the mailbox is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// create a FIFO box
Box* pBox = new Box("mybox",NU_FIFO);

UNSIGNED message[4];
// send a message on the given box. Suspend
// until it can be delivered

if ( pBox->Send(&message, NU_SUSPEND) == NU_SUCCESS )
{
    // we successfully delivered the message
}
```



Box::UpdateInfo

```
inline
BoxInfo&
Box::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox information service.
Post-condition	The info member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
aValidReference	Reference contains current information.

Example

```
// assume Box* pBox was created elsewhere

// get information on the box from the kernel
const BoxInfo& info = pBox->UpdateInfo();

// now examine the info to see if there is currently
// a message available
if ( info.IsMessage() )
{
    // yes, there is a message available...
```



class BoxInfo : public SuspendInfo

BoxInfo is a derived SuspendInfo class that holds various information about Box objects. NucleusPlusInfo objects hold various information about Nucleus C++ objects.

All Nucleus C++ Box objects within the system have a data member that is a pointer to an object of this information

Public Member Function

Member	Overview
~BoxInfo	Destructor
BoxInfo	Constructor
GetControlBlock	Returns a const reference to the Nucleus PLUS mailbox control block for the Box this BoxInfo object is associated with.
IsMessage	Returns true if an object is present in the Box object associated with this information object.
Update	Updates the information object for the Box object.



BoxInfo::~~BoxInfo

virtual

BoxInfo::~~BoxInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



BoxInfo::BoxInfo

BoxInfo::BoxInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// this object doesn't take any parameters, and is
// thus pretty unexciting. One of these objects
// exists inside of each Box object as a data member,
// and as such you shouldn't need to explicitly
// create one.
```

```
BoxInfo* pBoxInfo = new BoxInfo;
```



BoxInfo::GetControlBlock

```
inline
NU_MAILBOX&
BoxInfo::GetControlBlock()
const;
```

Returns a const reference to the Nucleus PLUS mailbox control block for the Box this BoxInfo object is associated

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidReference	const reference to the Nucleus PLUS mailbox control block for the Box this BoxInfo object is associated with.

Example

```
// assume Box mybox exists.

// get updated info from the kernel on the box
const BoxInfo& info = mybox.UpdateInfo();

// get a reference to the control box
const NU_MAILBOX& mcb = info.GetControlBlock();
```



BoxInfo::IsMessage

```
inline
BOOL
BoxInfo::IsMessage()
const;
```

Returns true if an object is present in the Box object associated with this information object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	FALSE if an object is NOT present in the Box object associated with this information object.
TRUE	TRUE if an object is present in the Box object associated with this information object.

Example

```
/ assume Box mybox exists.

// get updated info from the kernel on the box
const BoxInfo& info = mybox.UpdateInfo();

// check to see if there is a message in the box
BOOL bMessageIsAvailable = info.IsMessage();
```



BoxInfo::Update

virtual

STATUS

BoxInfo::Update();

Updates the information object for the Box object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's mailbox information service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume MyDerivedBox* pBox exists

// get a reference to the box's info object and
// assign it to our info object
BoxInfo info = pBox->UpdateInfo();

// sleep for a second
Task::Sleep( systemClockFrequency );

// update the info on the box
info.Update();

// get the number of tasks that are waiting for
// messages on the box
UNSIGNED tasks_waiting = info.GetNumberTasksWaiting();
```



class CommunicationInfo : public SuspendInfo

Holds various information about communication objects that hold multiple objects. This includes instances of both `Q` and `Pipe` classes.

Public Member Functions

Member	Overview
<code>~CommunicationInfo</code>	Destructor
<code>CommunicationInfo</code>	Constructor
<code>GetAvailable</code>	Returns the number of available data elements left in the container.
<code>GetBufferAddress</code>	Returns the starting address of the data memory.
<code>GetCurrent</code>	Number of data elements currently in the container.
<code>GetMessageSize</code>	Returns the number of bytes in each object.
<code>GetMessageType</code>	Returns the type of object supported in the container.
<code>GetSize</code>	Returns the number of data elements the container holds.



CommunicationInfo::~~CommunicationInfo

virtual

CommunicationInfo::~~CommunicationInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



CommunicationInfo::CommunicationInfo

CommunicationInfo::CommunicationInfo();

Constructor. `CommunicationInfo` is used as a common base class for information that is common to the different types of communication objects in Nucleus C++, namely pipes and queues.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

Not directly used in Nucleus C++.



CommunicationInfo::GetAvailable

```
inline
UNSIGNED
CommunicationInfo::GetAvailable()
const;
```

Returns the number of available data elements left in the container.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of available data elements left in the container.

Example

```
// assume that Q myQ exists.

// get current information on the Q from the kernel.
const QInfo& info = myQ.UpdateInfo();

// get the bytes available in the Q
UNSIGNED available = info.GetAvailable();
```



CommunicationInfo::GetBufferAddress

```
inline  
VOID*  
CommunicationInfo::GetBufferAddress()  
const;
```

Returns the starting address of the data memory.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A valid pointer to the starting address of the data memory.

Example

```
// assume that Q myQ exists.  
  
// get current information on the Q from the kernel.  
const QInfo& info = myQ.UpdateInfo();  
  
// get the bytes available in the Q  
VOID* pBuffer = info.GetBufferAddress();
```



CommunicationInfo::GetCurrent

```
inline
UNSIGNED
CommunicationInfo::GetCurrent()
const;
```

Number of data elements currently in the container.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of data elements currently in the container

Example

```
// assume that Pipe myPipe exists.

// get current info on the Pipe from the kernel.
const PipeInfo& info = myPipe.UpdateInfo();

// get the bytes available in the Pipe
UNSIGNED current_messages = info.GetCurrent();
```



CommunicationInfo::GetMessageSize

```
inline  
UNSIGNED  
CommunicationInfo::GetMessageSize()  
const;
```

Returns the number of bytes in each object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of bytes in each object

Example

```
// assume that Pipe myPipe exists.  
  
// get current info on the Pipe from the kernel.  
const PipeInfo& info = myPipe.UpdateInfo();  
  
// get the size of messages this pipe holds  
UNSIGNED message_byte_size = info.GetMessageSize();
```



CommunicationInfo::GetMessageType

```
inline
OPTION
CommunicationInfo::GetMessageType()
const;
```

Returns the type of object supported in the container.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validObject	The type of object supported in the container.

Example

```
// assume that Pipe myPipe exists.

// get current info on the Pipe from the kernel.
const PipeInfo& info = myPipe.UpdateInfo();

// get the type of messages this pipe holds
OPTION type = info.GetMessageType();

// type will be either NU_FIXED_SIZE or
// NU_VARIABLE_SIZE
```



CommunicationInfo::GetSize

```
inline  
UNSIGNED  
CommunicationInfo::GetSize()  
const;
```

Returns the number of data elements the container holds.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of data elements the container holds.

Example

```
// assume that Pipe myPipe exists.  
  
// get current info on the Pipe from the kernel.  
const PipeInfo& info = myPipe.UpdateInfo();  
  
// get the total number of bytes in the pipe  
UNSIGNED size = info.GetSize();
```



class DevelopmentService

DevelopmentService is a class that interfaces into the kernel's system development services. There are no data members and only static routines. This object can be used without declaring any objects of this class since there is only one development service system in the system.

Public Member Functions

Member	Overview
~DevelopmentService	Destructor
DevelopmentService	Constructor

Public Class Member Functions

Member	Overview
DisableHistorySaving	Disables history saving.
EnableHistorySaving	Enables history saving.
LicenseInformation	A global C string that contains customer license information.
MakeHistoryEntry	Allows users to make entries into the Nucleus PLUS circular log of various system activities.
ReleaseInformation	A global C string that contains the current version and release of the Nucleus PLUS software.
RetrieveHistoryEntry	Allows users to retrieve entries in the log of system activities.



DevelopmentService::~~DevelopmentService

virtual

```
DevelopmentService::~~DevelopmentService();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



DevelopmentService::DevelopmentService

```
DevelopmentService::DevelopmentService();
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

This class is used through it's static members, and thus you won't ever create an instance of it.



DevelopmentService::DisableHistorySaving

static

inline

VOID

```
DevelopmentService::DisableHistorySaving();
```

Disables history saving.

Overview

Condition	Description
Pre-condition	None
Action	Disables the kernel's history saving.
Post-condition	History saving is disabled.

Parameters

None

Return Value

None

Example

```
// turn history saving off
DevelopmentService::DisableHistorySaving();
```



DevelopmentService::EnableHistorySaving

```
static
inline
VOID
DevelopmentService::EnableHistorySaving();
```

Enables history saving.

Overview

Condition	Description
Pre-condition	None
Action	Enables the kernel's history saving.
Post-condition	History saving is enabled.

Parameters

None

Return Value

None

Example

```
// turn history saving on
DevelopmentService::EnableHistorySaving();
```



DevelopmentService::LicenseInformation

```
static  
inline  
CHAR*  
DevelopmentService::LicenseInformation();
```

A global C string that contains customer license information, including the customer's serial number, is available.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validString	The customer license string.

Example

```
// retrieve license information and stream it  
// out stdio  
cout << DevelopmentService::LicenseInformation();
```



DevelopmentService::MakeHistoryEntry

```

static
inline
VOID
DevelopmentService::MakeHistoryEntry
(
    UNSIGNED parameter1,

    UNSIGNED parameter2,
    UNSIGNED parameter3
);

```

Allows users to make entries into the Nucleus PLUS circular log of various system activities.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's make history entry service.
Post-condition	The entry is made in the log if the history-log capability is enabled. Otherwise, the service does nothing.

Parameters

Parameter	Overview
parameter1	General UNSIGNED variables that are specific to the user's needs.
parameter2	General UNSIGNED variables that are specific to the user's needs.
parameter3	General UNSIGNED variables that are specific to the user's needs.

Return Value

None



Example

```
// we decide to make a history entry to help us
// debug our code. The parameters are arbitrary and
// can be whatever makes sense in your application.
// Here we make up some defines that you might have
// in your application

DevelopmentService::MakeHistoryEntry
(
    ADC_TASK,           // who had the error
    ADC_INPUT_CLIPPED,  // what the error was
    (UNSIGNED)atodCounts // the actual reading
);
```



DevelopmentService::ReleaseInformation

```
static
inline
CHAR*
DevelopmentService::ReleaseInformation();
```

A global C string that contains the current version and release of the Nucleus PLUS software is available. Examination of this string in the target system provides quick identification of the underlying Nucleus PLUS system.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validString	The release string is returned.

Example

```
// stream the release information for the kernel
cout << "Nucleus PLUS version information"
      << DevelopmentService::ReleaseInformation();
```



DevelopmentService::RetrieveHistoryEntry

```
static
DevelopmentService
DevelopmentService::RetrieveHistoryEntry
(
    DATA_ELEMENT* id,

    UNSIGNED* parameter1,
    UNSIGNED* parameter2,
    UNSIGNED* parameter3,
    UNSIGNED* time,

    Task* task,
    HighLevelInterrupt* hisr
);
```

Allows users to retrieve entries in the Nucleus PLUS circular log of various system activities.

Overview

Condition	Description
Pre-condition	None
Action	Uses the retrieve history entry kernel service.
Post-condition	None

Parameters

Parameter	Overview
id	A pointer to a variable to hold the ID of the entry.
parameter1	Pointers to the history parameter's entries.
parameter2	Pointers to the history parameter's entries.
parameter3	Pointers to the history parameter's entries.
timeA	Pointer to a variable that is to hold the system clock portion of the entry.
TaskA	Pointer to a Task pointer to receive the Task that made the entry.
HisrA	Pointer to a HighLevelInterrupt pointer to receive the HighLevelInterrupt that made the entry.



Return Value

Return Value	Overview
NU_END_OF_LOG	Indicates that there are no more entries in the log.
NU_SUCCESS	Indicates successful completion of the service.

Example

```

// local variables
STATUS          status;
DATA_ELEMENT    id;
UNSIGNED        parameter1,parameter2,parameter3;
UNSIGNED        time;
Task*           pTask = NULL;
HighLevelInterrupt* pHisr = NULL;

// loop through the history log and stream all
// information out cout
while (NU_SUCCESS ==
    DevelopmentService::RetrieveHistoryEntry
    (
        &id,&parameter1,&parameter2,&parameter3,
        &time,&pTask,&pHisr
    ))
{
    //you got a good entry
    cout << "id" << id
        << "param1 " << parameter1
        << "param2 " << parameter2
        << "param3 " << parameter3
        << "time"    << parameter1;

    if (pTask != NULL)
    {
        const TaskInfo& info = pTask->UpdateInfo();
        cout << "Task " << info.GetName();
    }

    if (pHisr != NULL)
    {
        const HighLevelInterruptInfo& info =
            pTask->UpdateInfo();

        cout << "Hisr " << info.GetName();
    }
}

```



class Event

An Event object provides a mechanism to indicate that a certain event has occurred. An Event may be set, queried, and automatically cleared after it is sensed.

Public Member Functions

Member	Overview
~Event	Destructor
Clear	Clears this Event object.
Event	Constructor
Retrieve	Retrieves this Event object.

Public Class Member Functions

Member	Overview
initialize	One time class initialization called by the framework.
set	Sets this Event object.



Event::~Event

virtual

Event::~Event();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Removes the object from its associated event group container.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Event::Clear

inline

STATUS

```
Event::Clear();
```

Clears this Event object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's clear event service.
Post-condition	If successful, the event is cleared.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.
NU_INVALID_OPERATION	Indicates the operation parameter is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Event*  buttonEvent exists

// if the buttonEvent event is set, it
// will be cleared here.
buttonEvent->Clear();
```



Event::Event

Event::Event ();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Adds the object to its associated event group container.
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// create an event in the global heap
Event* myEvent = new Event;
```



Event::Initialize

static

STATUS

```
Event::Initialize();
```

One time class initialization called by the framework.

Overview

Condition	Description
Pre-condition	None
Action	Creates essential data elements to support internal Event class implementation details.
Post-condition	The Event class is initialized and objects of the class Event can be created.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.
NU_UNAVAILABLE	Indicates the semaphore is unavailable.

Example

```
// This member is called by the framework once on
// initialization, you shouldn't have to call it.

if ( NU_SUCCESS == Event::Initialize() )
{
    ...
}
```



Event::Retrieve

```

inline
STATUS
Event::Retrieve
(
    BOOL consume = TRUE,
    UNSIGNED suspend = NU_SUSPEND
);

```

Retrieves this Event object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's retrieve event service.
Post-condition	If successful and the consume input parameter is TRUE, the event will be automatically cleared upon exit.

Parameters

Parameter	Overview
consume	If is TRUE (default action) the event is automatically cleared on a successful request.
suspend	Specifies how the calling task is to suspend if the request cannot be immediately satisfied. By default, tasks suspend on the service.

Return Value

Return Value	Overview
NU_GROUP_DELETED	Event flag group was deleted while the task was suspended.
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.
NU_INVALID_OPERATION	Indicates the operation parameter is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_NOT_PRESENT	Indicates the requested event flag combination is not currently present.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates the requested event flag combination is not present even after the specified suspension timeout.



Example

```
// assume that Event*  buttonEvent exists

STATUS status;

// example 1:
// retrieve the button event and specify that the
// event is automatically cleared for next time by
// our retrieval.  Use default parameter for
// suspension (which says to suspend indefinitely)

    status = buttonEvent->Retrieve( TRUE );

    if (status == NU_SUCCESS)
    {
        // you got a button press, do your thing here.
        ...

// example 2:
// retrieve the button event and specify that we
// don't want to clear the event (other tasks may
// need it as well, and it is someone elses
// responsibility to clear the event).  Also specify
// that we want to suspend waiting for the event
// for a maximum of 20 clock ticks.

    status = buttonEvent->Retrieve( FALSE, 20 );

    if (status == NU_TIMEOUT)
    {
        // you timed out waiting for the button
        // press event...
```



Event::Set

```
static
inline
STATUS
Event::Set () ;
```

Sets this Event object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's set event service.
Post-condition	If successful, the event is set.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.
NU_INVALID_OPERATION	Indicates the operation parameter is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// in this example, we assume that we have a high
// level interrupt class that is activated by
// button presses and then sets an event that
// that we want to suspend waiting for the event// some sort of user
// interface task would be
// for a maximum of 20 clock ticks.// suspended on.

status = buttonEvent->Retrieve( FALSE, 20 );void ButtonHISR::Entry()
{
    if (status == NU_TIMEOUT) // assume we have a member Event*
```



class EventGroup : public NucleusPlus

The Nucleus C++ PLUS event classes are used to create real-time architectures that are event-driven, providing a commonly required synchronization mechanism. The Nucleus C++ classes are `Event` and `EventGroup`. Event groups support the ability to suspend waiting for a logical combination of multiple events to occur.

Public Member Functions

Member	Overview
<code>~EventGroup</code>	Destructor
<code>EventGroup</code>	Constructor
<code>EventGroup</code>	Constructor
<code>Retrieve</code>	Retrieves the specified event flag combination from this event group.
<code>Set</code>	Sets the specified event flags in this group.
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the info object.

class EventGroupInfo : public SuspendInfo

Object that holds various information on an `EventGroup` object.

Public Member Functions

Member	Overview
<code>~EventGroupInfo</code>	Destructor
<code>EventGroupInfo</code>	Constructor
<code>GetControlBlock</code>	Returns a const reference to the Nucleus PLUS event group control block for the <code>EventGroup</code> this <code>EventGroupInfo</code> object is associated with.
<code>GetFlags</code>	Returns the current flags in the event group.
<code>Update</code>	Updates the information object for the <code>EventGroup</code> .



EventGroupInfo::~EventGroupInfo

virtual

EventGroupInfo::~EventGroupInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



EventGroupInfo::EventGroupInfo

EventGroupInfo::EventGroupInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that EventGroup* pButtonEvents exists.

// create and EventGroupInfo object.
EventGroupInfo info;

// assign it information from the pButtonEvents object
```



EventGroupInfo::GetControlBlock

inline

NU_EVENT_GROUP&

EventGroupInfo::GetControlBlock()

const;

Returns a const reference to the Nucleus PLUS event group control block for the EventGroup this EventGroupInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidReference	const reference to the Nucleus PLUS event group control block for the EventGroup this EventGroupInfo object is associated with.



EventGroupInfo::GetFlags

```
inline  
UNSIGNED  
EventGroupInfo::GetFlags()  
const;
```

Returns the current flags in the event group.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current flags in the event group.

Example

```
// assume that EventGroup* pEvents exists.  
  
const EventGroupInfo& info = pEvents->UpdateInfo();  
  
// return the current bitmask of the event group.  
UNSIGNED flags = info.GetFlags();
```



EventGroupInfo::Update

virtual

STATUS

EventGroupInfo::Update();

Updates the information object for the EventGroup.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's event group information service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that EventGroup* pButtonEvents exists.

// create and EventGroupInfo object.
EventGroupInfo info;

// assign it information from the pButtonEvents object
info = pButtonEvents->UpdateInfo();

...

// get the latest information from the kernel.
info.Update();
```



class HighLevelInterrupt : public NucleusPlus

The Nucleus C++ PLUS classes `LowLevelInterrupt` and `HighLevelInterrupt` are base classes that manage asynchronous external interrupts in an embedded application. The classes demonstrate the power of *portability*. The various interrupt architectures found in today's advanced embedded processors are very different but the Nucleus C++ PLUS classes do not across all supported Nucleus C++ embedded processor targets.

Public Member Functions

Member	Overview
<code>~HighLevelInterrupt</code>	Destructor
<code>Activate</code>	Activates the <code>HighLevelInterrupt</code> .
<code>HighLevelInterrupt</code>	Constructor
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the info object.

Protected Member Functions

Member	Overview
<code>Entry</code>	<code>Entry()</code> is the entry routine for derived <code>HighLevelInterrupt</code> classes. This is the member that is executed by the kernel when a HISR is activated.

Public Class Member Functions

Member	Overview
<code>Current</code>	Returns a pointer to the current executing <code>HighLevelInterrupt</code> object.
<code>GetHighLevelInterrupt</code>	Returns a pointer to the <code>HighLevelInterrupt</code> object associated with a particular kernel HISR control block. parameters.



HighLevelInterrupt::~~HighLevelInterrupt

virtual

HighLevelInterrupt::~~HighLevelInterrupt();

Destructor

Overview

Condition	Description
Pre-condition	None.
Action	Uses the kernel's delete HISR service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



HighLevelInterrupt::Activate

inline

STATUS

HighLevelInterrupt::Activate()

Activates the HighLevelInterrupt. If the specified HighLevelInterrupt is currently executing, this activation request is not processed until the current execution is complete. A HighLevelInterrupt is activated once for each activation request.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's activate HISR service.
Post-condition	A HighLevelInterrupt is activated and scheduled for execution in the kernel.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_HISR	Indicates the HISR pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// in this example, we have a low level interrupt
// that is activated by an hardware interrupt
// signifying that the user has pressed a button.
// we just activate a High Level Interrupt (that is
// a member of the low level interrupt) to handle
// the button press.

void ButtonLowLevelInterrupt::Entry()
{
    // activate the HISR
    buttonHISR.Activate();
}
```



HighLevelInterrupt::Current

```
static
inline
HighLevelInterrupt*
HighLevelInterrupt::Current();
```

Returns a pointer to the current executing HighLevelInterrupt object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's current HISR service.
Post-condition	None

Parameters

None

Return Value

Return Value	Description
aValidPointer	A valid pointer to the current executing HighLevelInterrupt object.

Example

```
HighLevelInterrupt* pCurrent = HighLevelInterrupt::Current();

if (pCurrent != NULL)
{
    // there was a High Level interrupt running...
```



HighLevelInterrupt::Entry

virtual

VOID

HighLevelInterrupt::Entry()=0;

Entry() is the entry routine for derived HighLevelInterrupt classes. This is the member that is executed by the kernel when a HISR is activated.

Overview

Condition	Description
Pre-condition	None.
Action	Derived class specific behavior. HOWEVER, derived class implementations must return before any task scheduling can occur in the kernel. Remember, this is a class representing an interrupt service routine.
Post-condition	The execution activation counter is decremented.

Parameters

None

Return Value

None

Example

```
// This routine is called by the framework for you
// when Activate is called for your HISR object.

void MyHighLevelInterrupt::Entry()
{
    panicAndRunEvent.Set(); // do your thing here
}
```



HighLevelInterrupt::GetHighLevelInterrupt

```

static
inline
HighLevelInterrupt*
HighLevelInterrupt::GetHighLevelInterrupt
(
    NU_HISR*
    nuhistr
);

```

Returns a pointer to the HighLevelInterrupt object associated with a particular kernel HISR control block.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

Parameter	Overview
nuhistr	A pointer to the Nucleus PLUS HISR control block.

Return Value

Return Value	Overview
aValidPointer	A valid pointer to the HighLevelInterrupt object associated with a particular kernel HISR control block.

Example

```

// assume that somewhere is the High Level
// Interrupt control block: NU_HISR hcb;

HighLevelInterrupt* pHisrObject =
HighLevelInterrupt::GetHighLevelInterrupt( &hcb );

```



HighLevelInterrupt::HighLevelInterrupt

HighLevelInterrupt

HighLevelInterrupt::HighLevelInterrupt

```
(  
    const CHAR* name,  
    OPTION  
    priority,  
    UNSIGNED stacksize,  
    CHAR* stack = NULL  
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create HISR service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The name of the object.
priority	The priority (0-2).
stacksize	Number of bytes in stack.
stack	If supplied, the hisr will use the area pointed to as the stack for this hisr object. Otherwise, the stack is allocated from the free store.

Return Value

None



Example

```
// here is the declaration of your own derived
// high level interrupt class

#include "npp.h"

class ButtonHISR : public HighlevelInterrupt
{
    public:
        ButtonHISR();
        virtual ~ButtonHISR();

    protected:
        virtual void Entry();
};

// here we create a derived HighLevelInterrupt object
// with the parameters below. Note that for the
// location of the stack we passed in NULL, which
// means that the HIsr will allocate it from the
// global C++ free store.
```



HighLevelInterrupt::UpdateInfo

```
inline
HighLevelInterruptInfo&
HighLevelInterrupt::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's event group information service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	Reference contains current information.

Example

```
// get updated information on the object
// from the kernel.

const HighLevelInterruptInfo& info =
    // do any specific construction herebuttonHisr.UpdateInfo();
}
```



class HighLevelInterruptInfo : public NucleusPlusInfo

Derived NucleusPlusInfo class that holds various information about a HighLevelInterrupt object.

Public Member Functions

Member	Overview
~HighLevelInterruptInfo	Destructor
GetControlBlock	Returns a const reference to the Nucleus PLUS high level interrupt control block for the HighLevelInterrupt this HighLevelInterruptInfo object is associated with.
HighLevelInterruptInfo	Constructor
Update	Updates the information object for the HighLevelInterrupt.



HighLevelInterruptInfo::~HighLevelInterruptInfo

virtual

```
HighLevelInterruptInfo::~HighLevelInterruptInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



HighLevelInterruptInfo::GetControlBlock

```
inline
NU_HISR&
HighLevelInterruptInfo::GetControlBlock()
const;
```

Returns a const reference to the kernel's high level interrupt control block for the HighLevelInterrupt this HighLevelInterruptInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

aValidReference const reference to the Nucleus PLUS high level interrupt control block for the HighLevelInterrupt this HighLevelInterruptInfo object is associated with.

Example

```
// assume that const HighLevelInterruptInfo& info
// exists.
```

```
const NU_HISR& h_h = info.GetControlBlock();
```



HighLevelInterruptInfo::HighLevelInterruptInfo

HighLevelInterruptInfo::HighLevelInterruptInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Constructor for HighLevelInterruptInfo objects.
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that derived HighLevelInterrupt
// buttonHsr exists.

HighLevelInterruptInfo info;
```



HighLevelInterruptInfo::Update

virtual

STATUS

HighLevelInterruptInfo::Update();

Updates the information object for the HighLevelInterrupt.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's high level interrupt information service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_HISR	Indicates the HISR pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that HighLevelInterrupt* buttonHsr exists.

// create and info object.
HighLevelInterruptInfo info;

// assign it information from the object
info = buttonHsr->UpdateInfo();
```



class InterruptSystem

The `InterruptSystem` class is a static object that controls the system interrupts. There are no data members and only static routines. This object can be used without declaring any objects of this class.

Public Member Functions

Member	Overview
<code>~InterruptSystem</code>	Destructor
<code>InterruptSystem</code>	Constructor

Public Class Member Functions

Member	Overview
<code>Disable</code>	Disables interrupts in a task-independent manner.
<code>Enable</code>	Enables interrupts.
<code>SetupVector</code>	Replaces the interrupt vector specified with the custom interrupt service routine supplied.



InterruptSystem::~~InterruptSystem

virtual

InterruptSystem::~~InterruptSystem();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



InterruptSystem::Disable

static

inline

INT

```
InterruptSystem::Disable();
```

Disables interrupts in a task-independent manner. An interrupt disabled by this service remains disabled until enabled by a subsequent call to `Enable`.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's disable interrupt service.
Post-condition	Interrupts are disabled in the system.

Parameters

None

Return Value

Return Value	Overview
validData	The previous level of enabled interrupts.

Example

```
InterruptSystem::Disable();

// do whatever was critical to have interrupts off
// here
```



InterruptSystem::Enable

```
static
inline
INT
InterruptSystem::Enable();
```

Enables interrupts in a task-independent manner.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's enable interrupt service.
Post-condition	Interrupts are disabled in the system.

Parameters

None

Return Value

Return Value	Overview
validData	The previous level of enabled interrupts.

Example

```
InterruptSystem::Disable();

// do whatever was critical to have interrupts off
// here
```



InterruptSystem::InterruptSystem

```
InterruptSystem::InterruptSystem();
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

This object is used through static members and is never explicitly created.



InterruptSystem::SetupVector

```

static
inline
VOID*
InterruptSystem::SetupVector
(
    INT vector,
    VOID* isr
);

```

Replaces the interrupt vector specified with the custom interrupt service routine supplied. The previous interrupt vector contents are returned by the service. The interrupt service routine is responsible for saving and restoring all registers

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's setup vector service.
Post-condition	The interrupt vector has been replaced with a pointer to the <code>isr</code> supplied.

Parameters

Parameter	Overview
<code>vector</code>	The interrupt vector to be replaced.
<code>isr</code>	The new Interrupt Service Routine for the vector.



Return Value

Return Value	Overview
AValidPointer	The previous interrupt vector contents.

Example

```
void MyISR()
{
    ...
}

...

// pointer to old ISR function
void (*pISR)();

// return vector 10 to 15 to previous function
```



class LowLevelInterrupt

The Nucleus C++ PLUS classes `LowLevelInterrupt` and `HighLevelInterrupt` are base classes that manage asynchronous external interrupts in an embedded application. The classes demonstrate the power of *portability*. The various interrupt architectures found in today's advanced embedded processors are very different but the Nucleus C++ PLUS classes do not across all supported Nucleus C++ embedded processor targets.

Public Member Functions

Member	Overview
<code>~LowLevelInterrupt</code>	Destructor
<code>Entry</code>	Derived <code>LowLevelInterrupt</code> classes will define a virtual <code>Entry()</code> member with specific behavior. This member is called when the LISR interrupt is fired.
<code>GetVector</code>	Returns the vector for this LISR.
<code>LowLevelInterrupt</code>	Constructor

Public Class Member Functions

Member	Overview
<code>Initialize</code>	One time class initialization called by the framework.



LowLevelInterrupt::~LowLevelInterrupt

virtual

LowLevelInterrupt::~LowLevelInterrupt();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's unregister LISR service. Subsequent object instantiations for the same interrupt vector will cause old objects to be replaced. When the object that replaces an old object is destructed, the old object is reinstalled. Multiple replacements can occur and old objects will be reinstalled on a last-replaced-first-reinstalled basis. That is to say, the last object that was replaced will be the first object to be reinstalled. If old objects that have been replaced are destructed, they are simply removed from the list of old objects, and therefore will not be reinstalled.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



LowLevelInterrupt::Entry

virtual

VOID

LowLevelInterrupt::Entry()=0;

Derived LowLevelInterrupt classes will define a virtual Entry() member with specific behavior. This member is called when the LISR interrupt is fired.

Overview

Condition	Description
Pre-condition	Derived class specific.
Action	Derived class specific.
Post-condition	Derived class specific.

Parameters

None

Return Value

None

Example

```
// the framework calls this member when an
// interrupt fires.

void MyLISR::Entry()
{
    // called when whatever vector we registered
    // this interrupt for is triggered.
}
```



LowLevelInterrupt::GetVector

```
inline  
INT  
LowLevelInterrupt::GetVector()  
const;
```

Returns the vector for this LISR.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validObject	The vector for this LISR.

Example

```
// in this example, we have a High Level interrupt  
// that has a low level interrupt as a member.  
// when it is activated, it checks to see what  
// vector the Low level interrupt is using.  
  
void MyCommHISR::Entry()  
{  
    INT vector = myCommLisr.GetVector();  
    ...  
}
```



LowLevelInterrupt::Initialize

static

STATUS

LowLevelInterrupt::Initialize();

One time class initialization called by the framework.

Overview

Condition	Description
Pre-condition	None
Action	Creates and initializes internal data elements.
Post-condition	LowLevelInterrupt objects are initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.
NU_UNAVAILABLE	Indicates the semaphore is unavailable.

Example

This member is called by the framework.



LowLevelInterrupt::LowLevelInterrupt

LowLevelInterrupt::LowLevelInterrupt

```
(
    INT lisrVector
);
```

Constructor. **WARNING:** interrupts for the supplied vector cannot occurring during this initialization member. If this is possible, please disable interrupts during execution of the constructor.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's register LISR service. Subsequent object instantiations for the same interrupt vector will cause old objects to be replaced. When the object that replaces an old object is destructed, the old object is reinstalled. Multiple replacements can occur and old objects will be reinstalled on a last-replaced-first-reinstalled basis. That is to say, the last object that was replaced will be the first object to be reinstalled. If old objects that have been replaced are destructed, they are simply removed from the list of old objects, and therefore will not be reinstalled.
Post-condition	The object exists and the low level interrupt object is registered. NOTE: interrupts for the supplied vector can now safely occur.

Parameters

Parameter	Overview
lisrVector	The interrupt vector associated with the object.

Return Value

None



Example

```
// in this example, we show an example
// constructor for a derived LowLevelInterrupt
// class

class MyButtonInterrupt : public LowLevelInterrupt
{
    public:

        MyButtonInterrupt( INT vector );
        virtual ~MyButtonInterrupt();

    protected:
        virtual void Entry();
};
```



class `MemoryPool` : public `NucleusPlus`

The Nucleus C++ BASE package includes a re-entrant, real-time solution for the standard C++ memory operators `new` and `delete`. Nucleus C++ PLUS extends this and adds classes `MemoryPool` and `PartitionPool`. These classes encapsulate Nucleus PLUS memory management services.

Public Member Functions

Member	Overview
<code>~MemoryPool</code>	Destructor
<code>Allocate</code>	Allocates a block of memory from this <code>MemoryPool</code> object.
<code>Deallocate</code>	Returns the block of memory to this <code>MemoryPool</code> object.
<code>MemoryPool</code>	Constructor
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the <code>info</code> object.



MemoryPool::~MemoryPool

virtual

MemoryPool::~MemoryPool();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete memory pool service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



MemoryPool::Allocate

```
inline
STATUS
MemoryPool::Allocate
(
    VOID* returnPointer,
    UNSIGNED size,
    UNSIGNED
suspend = NU_SUSPEND
);
```

Allocates a block of memory from this MemoryPool object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's allocate memory pool service.
Post-condition	If successful, the pool has less memory.

Parameters

Parameter	Overview
returnPointer	The destination of the returned pointer for the service. Points to the first byte of the allocated memory.
size	The number of bytes to allocate.
suspend	Specifies how tasks suspend on the request.

Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_POOL	Indicates the memory pool pointer is invalid.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_NO_MEMORY	Indicates the memory request could not be immediately satisfied.
NU_POOL_DELETED	Memory pool was deleted while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates the requested memory is still unavailable even after suspending for the specified timeout value.



Example

```
// assume that MemoryPool mypool exists.

// Allocate a block 256 bytes big from the
// mypool memory pool.

VOID* pBlock;
STATUS status;

status = myPool.Allocate(&pBlock,256,NU_SUSPEND);

if (status == NU_SUCCESS)
{
    ...
}
```



MemoryPool::Deallocate

```
inline
STATUS
MemoryPool::Deallocate
(
    VOID* memory
);
```

Returns the block of memory to this MemoryPool object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's deallocate memory pool service.
Post-condition	If successful, the pool has more memory.

Parameters

Parameter	Overview
memory	Points to the block of memory to return.

Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that MemoryPool mypool exists.

// Allocate a block 256 bytes big from the
// mypool memory pool.

VOID* pBlock;
STATUS status;

status = myPool.Allocate(&pBlock, 256, NU_SUSPEND);
```



MemoryPool::MemoryPool

```
MemoryPool::MemoryPool
(
const CHAR* name,
    UNSIGNED size,
    UNSIGNED minimumAllocation,

    OPTION suspendType,
    VOID* startAddress = NULL
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create memory pool service.
Post-condition	The object exists.

Parameters

Parameter	Overview
Name	The name of MemoryPool.
Size	The size of Memory Pool.
MinimumAllocation	The minimum number of bytes in each allocation.
SuspendType	Specifies how tasks suspend on an object.
StartAddress	An optional starting address for a pool. A standard operating system allocation is performed and a starting address assigned if the parameter is absent or zero.

Return Value

None



Example

```
// example 1: Create a memory pool named bob,  
// 1 meg big, 1024 byte minimum allocation size,  
// and allocate the pool from the heap.  
  
MemoryPool myPool("bob",1048576,1024,NULL);  
  
// example 1: Create a memory pool named vern,  
// 1 meg big, 256 byte minimum allocation size,  
// and allocate the pool at address 0x1FFF0000  
  
MemoryPool  
myPool("vern",1048576,256,(VOID*)0x1FFF0000);
```



MemoryPool::UpdateInfo

```
inline
MemoryPoolInfo&
MemoryPool::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's memory pool information service.
Post-condition	The info member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	A reference to the info object.

Example

```
// tell the object to update the information, and
// then give us a const reference to the info
// object
```



class `MemoryPoolInfo` : public `PoolInfo`

Object that holds various information on a `MemoryPool` object.

Public Member Functions

Member	Overview
<code>~MemoryPoolInfo</code>	Destructor
<code>GetControlBlock</code>	Returns a <code>const</code> reference to the kernel's memory pool control block for the <code>MemoryPool</code> this <code>MemoryPoolInfo</code> object is associated with.
<code>GetMinimumAllocation</code>	Returns the minimum number of bytes for each allocation in the memory pool.
<code>MemoryPoolInfo</code>	Constructor
<code>Update</code>	Updates the information object for the <code>MemoryPool</code> .



MemoryPoolInfo::~MemoryPoolInfo

virtual

MemoryPoolInfo::~MemoryPoolInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



MemoryPoolInfo::GetControlBlock

```
inline  
NU_MEMORY_POOL&  
MemoryPoolInfo::GetControlBlock()  
const;
```

Returns a `const` reference to the kernel's memory pool control block for the `MemoryPool` this `MemoryPoolInfo` object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Description
<code>AValidReference</code>	<code>const</code> reference to the kernel's memory pool control block for the <code>MemoryPool</code> this <code>MemoryPoolInfo</code> object is associated with.

Example

```
// assume that MemoryPool* pCommPool exists.  
  
const MemoryPoolInfo& info = pCommPool->UpdateInfo();
```



MemoryPoolInfo::GetMinimumAllocation

```
inline
UNSIGNED
MemoryPoolInfo::GetMinimumAllocation()
const;
```

Returns the minimum number of bytes for each allocation in the memory pool.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The minimum number of bytes for each allocation in the memory pool.

Example

```
// assume that MemoryPool* pCommPool exists.

const MemoryPoolInfo& info = pCommPool->UpdateInfo();
```



MemoryPoolInfo::MemoryPoolInfo

MemoryPoolInfo::MemoryPoolInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that MemoryPool* pCommPool exists.  
  
MemoryPoolInfo info;
```



MemoryPoolInfo::Update

virtual

STATUS

MemoryPoolInfo::Update();

Updates the information object for the MemoryPool.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's memory pool information service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_POOL	Indicates the memory pool pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that MemoryPool* pPool exists.

// create and info object.
MemoryPoolInfo info;

// assign it information from the MemoryPool object
info = pPool->UpdateInfo();

...

// get the latest information from the kernel.
info.Update();
```



class NppPLUS : public NppComponent

The Nucleus C++ PLUS component class manages the class interface into the Nucleus PLUS RTOS. It is required to create and initialize an instance of this Nucleus C++ component prior to using any class interfaces.

Public Member Functions

Member	Overview
~NppPLUS	Destructor
Initialize	This virtual member initializes the NppPLUS component.
NppPLUS	Constructor

Protected Member Functions

Member	Overview
InitializeBox	Initialize the Box class.
InitializeEvent	Initializes the Event class.
InitializeEventGroup	Initialize EventGroup objects.
InitializeHelpers	Initialize helpers.
InitializeHighLevelInterrupt	Initialize HighLevelInterrupt objects.
InitializeIODriver	Initialize IODriver objects.
InitializeLowLevelInterrupt	Initialize LowLevelInterrupt objects.
InitializeMemoryPool	Initialize memory pool objects.
InitializePartitionPool	Initialize partition pool objects.
InitializePipe	Initialize Pipe objects.
InitializeQ	Initialize Q objects.
InitializeSemaphore	Initialize Semaphore objects.
InitializeTask	Initialize Task objects.
InitializeTimer	Initialize Timer objects.



NppPLUS : : ~NppPLUS

virtual

NppPLUS : : ~NppPLUS () ;

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NppPLUS::Initialize

```
virtual  
STATUS  
NppPLUS::Initialize();
```

This virtual member initializes the NppPLUS component. Final initialization occurs after the object completely exists. This is required since NppPLUS uses protected class callback methods and C++ does not allow virtual functions to be called within the constructor of a class.

Overview

Condition	Description
Pre-condition	None
Action	Initializes each Nucleus C++ PLUS class.
Post-condition	The Nucleus C++ PLUS class interface is available for use.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
STATUS status;  
  
// create the component  
NppPLUS* nppPLUS = new NppPLUS;  
  
// initialize the component
```



NppPLUS::InitializeBox

virtual

STATUS

NppPLUS::InitializeBox();

Initialize the Box class.

Overview

Condition	Description
Pre-condition	None
Action	Calls the Box class Initialize member.
Post-condition	The Box class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeBox();
    ...
}
```



NppPLUS::InitializeEvent

virtual

STATUS

NppPLUS::InitializeEvent();

Initializes the Event class.

Overview

Condition	Description
Pre-condition	None
Action	Calls the <code>Event</code> class <code>Initialize</code> member.
Post-condition	The <code>Event</code> class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeEvent();
    ...
}
```



NppPLUS::InitializeEventGroup

virtual

STATUS

NppPLUS::InitializeEventGroup();

Initialize EventGroup objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the EventGroup class Initialize member.
Post-condition	The EventGroup class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeEventGroup();
    ...
}
```



NppPLUS::InitializeHelpers

virtual

STATUS

NppPLUS::InitializeHelpers();

Initialize helpers.

Overview

Condition	Description
Pre-condition	None
Action	If helpers are enabled in the application tuning file NPPAPP.H, the helper objects in the system are created.
Post-condition	If enabled, the helper tasks will exist.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    #if (NPP_HELPERS)
```



NppPLUS::InitializeHighLevelInterrupt

virtual

STATUS

NppPLUS::InitializeHighLevelInterrupt();

Initializes HighLevelInterrupt objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the HighLevelInterrupt class Initialize member.
Post-condition	The HighLevelInterrupt class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeHighLevelInterrupt();
    ...
}
```



NppPLUS::InitializeIODriver

virtual

STATUS

NppPLUS::InitializeIODriver();

Initialize IODriver objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the IODriver class Initialize member.
Post-condition	The IODriver class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeIODriver();
    ...
}
```



NppPLUS::InitializeLowLevelInterrupt

virtual

STATUS

NppPLUS::InitializeLowLevelInterrupt();

Initialize LowLevelInterrupt objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the LowLevelInterrupt class Initialize member.
Post-condition	The LowLevelInterrupt class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeLowLevelInterrupt();
    ...
}
```



NppPLUS::InitializeMemoryPool

virtual

STATUS

NppPLUS::InitializeMemoryPool();

Initialize MemoryPool objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the MemoryPool class Initialize member.
Post-condition	The MemoryPool class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeMemoryPool();
    ...
}
```



NppPLUS::InitializePartitionPool

virtual

STATUS

NppPLUS::InitializePartitionPool();

Initialize PartitionPool objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the PartitionPool class Initialize member.
Post-condition	The PartitionPool class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializePartitionPool();
    ...
}
```



NppPLUS::InitializePipe

virtual

STATUS

NppPLUS::InitializePipe();

Initialize Pipe objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the Pipe class Initialize member.
Post-condition	The Pipe class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializePipe();
    ...
}
```



NppPLUS::InitializeQ

virtual

STATUS

NppPLUS::InitializeQ();

Initialize Q objects.

Overview

Condition	Description
Pre-condition	None.
Action	Calls the Q class <code>Initialize</code> member.
Post-condition	The Q class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeQ();
    ...
}
```



NppPLUS::InitializeSemaphore

virtual

STATUS

NppPLUS::InitializeSemaphore();

Initialize Semaphore objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the Semaphore class Initialize member.
Post-condition	The Semaphore class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeSemaphore();
    ...
}
```



NppPLUS::InitializeTask

virtual

STATUS

NppPLUS::InitializeTask();

Initialize Task objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the Task class <code>Initialize</code> member.
Post-condition	The Task class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeTask();
    ...
}
```



NppPLUS::InitializeTimer

virtual

STATUS

NppPLUS::InitializeTimer();

Initialize Timer objects.

Overview

Condition	Description
Pre-condition	None
Action	Calls the Timer class Initialize member.
Post-condition	The Timer class is initialized.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	The initialization process was successful.

Example

```
// NppPLUS::Initialize is responsible for calling
// all of the individual object's static
// initialization

STATUS
NppPLUS::Initialize()
{
    ...
    InitializeTimer();
    ...
}
```



NppPLUS : :NppPLUS

NppPLUS : :NppPLUS () ;

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
NppPLUS* nppPLUS = new NppPLUS;
```



class NppProtect : public NucleusPlus

A Nucleus C++ protection class provides multithreaded protection of a critical data structure.

Public Member Functions

Member	Overview
~NppProtect	Destructor
GetProtectedBYTE_PTR	Returns the BYTE_PTR value stored in "bp", protecting the variable from multiple thread access using this protection object.
GetProtectedCHAR	Returns the CHAR value stored in "c", protecting the variable from multiple thread access using this protection object.
GetProtectedINT	Returns the INT value stored in "i", protecting the variable from multiple thread access using this protection object.
GetProtectedUNSIGNED	Returns the UNSIGNED value stored in "u", protecting the variable from multiple thread access using this protection object.
GetProtectedUNSIGNED_PTR	Returns the UNSIGNED_PTR value stored in "up", protecting the variable from multiple thread access using this protection object.
NppProtect	Constructor
Protect	Protects a data structure from access on another thread until the Unprotect member is called.
SetProtectedBYTE_PTR	Sets the BYTE_PTR value stored in "bp" with "newBp", protecting the variable from multiple thread access using this protection object.
SetProtectedCHAR	Sets the CHAR value stored in "c" with "newC", protecting the variable from multiple thread access using this protection object.
SetProtectedINT	Sets the INT value stored in "i" with "newI", protecting the variable from multiple thread access using this protection object.
SetProtectedUNSIGNED	Sets the UNSIGNED value stored in "u" with "newU", protecting the variable from multiple thread access using this protection object.
SetProtectedUNSIGNED_PTR	Sets the UNSIGNED_PTR value stored in "up" with "newUp", protecting the variable from multiple thread access using this protection object.
UnProtect	Releases protection of the data structure to other threads.



NppProtect::~NppProtect

virtual

NppProtect::~NppProtect();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NppProtect::GetProtectedBYTE_PTR

```
inline
BYTE_PTR
NppProtect::GetProtectedBYTE_PTR
(
    BYTE_PTR bp
);
```

Returns the BYTE_PTR value stored in "bp", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Obtains a protected copy of the BYTE_PTR.
Post-condition	None

Parameters

Parameter	Overview
bp	BYTE_PTR value.

Return Value

Return Value	Overview
validData	The BYTE_PTR value stored in "bp".

Example

```
// assume that NppProtect* pProtect exists.
// in this example, the data below could be modified
// by multiple threads, or hisrs, or I/O drivers. To
// make sure that we don't get half an updated
// value when accessing it, we have to protect
// access to it.

// elsewhere... BYTE_PTR volatile_data;

BYTE_PTR my_safe_data;
```



NppProtect::GetProtectedCHAR

```

inline
CHAR
NppProtect::GetProtectedCHAR
(
    CHAR c
);

```

Returns the CHAR value stored in "c", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Obtains a protected copy of the CHAR.
Post-condition	None

Parameters

Parameter	Overview
c	CHAR value

Return Value

Return Value	Overview
validData	The CHAR value stored in "c".

Example

```

// assume that NppProtect* pProtect exists.
// in this example, the data below could be modified
// by multiple threads, or hisrs, or I/O drivers. To
// make sure that we don't get half an updated
// value when accessing it, we have to protect
// access to it.

// elsewhere... CHAR volatile_data;
CHAR my_safe_data;

```



NppProtect::GetProtectedINT

```
inline
INT
NppProtect::GetProtectedINT
(
    INT i
);
```

Returns the `INT` value stored in "i", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Obtains a protected copy of <code>INT</code> .
Post-condition	None

Parameters

Parameter	Overview
i	<code>INT</code> value.

Return Value

Return Value	Overview
validData	The <code>INT</code> value stored in "i".

Example

```
// assume that NppProtect* pProtect exists.
// in this example, the data below could be modified
// by multiple threads, or hisrs, or I/O drivers. To
// make sure that we don't get half an updated
// value when accessing it, we have to protect
// access to it.
// elsewhere... INT volatile_data;
INT my_safe_data;
my_safe_data =
pProtect->GetProtectedINT( volatile_data );
```



NppProtect::GetProtectedUNSIGNED

```

inline
UNSIGNED
NppProtect::GetProtectedUNSIGNED
(
    UNSIGNED u
);

```

Returns the UNSIGNED value stored in "u", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Obtains a protected copy of UNSIGNED.
Post-condition	None

Parameters

Parameter	Overview
u	UNSIGNED value.

Return Value

Return Value	Overview
validData	The UNSIGNED value stored in "u".



Example

```
// assume that NppProtect* pProtect exists.
// in this example, the data below could be modified
// by multiple threads, or hisrs, or I/O drivers. To
// make sure that we don't get half an updated
// value when accessing it, we have to protect
// access to it. Yes, wether this is even an issue
// in your application, is a function of your
// architecture as well as alignment...
// elsewhere... UNSIGNED volatile_data;

UNSIGNED my_safe_data;

my_safe_data =
pProtect->GetProtectedUNSIGNED( volatile_data );

// now my_safe_data contains a good copy of the
// shared data.
```



NppProtect::GetProtectedUNSIGNED_PTR

```

inline
UNSIGNED_PTR
NppProtect::GetProtectedUNSIGNED_PTR
(
    UNSIGNED_PTR up
);

```

Returns the UNSIGNED_PTR value stored in "up", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Obtains a protected copy of UNSIGNED_PTR.
Post-condition	None

Parameters

Parameter	Overview
up	UNSIGNED_PTR value.

Return Value

Return Value	Overview
validData	The UNSIGNED_PTR value stored in "up".



Example

```
// assume that NppProtect* pProtect exists.  
// in this example, the data below could be modified  
// by multiple threads, or hisrs, or I/O drivers. To  
// make sure that we don't get half an updated  
// value when accessing it, we have to protect  
// access to it. Yes, wether this is even an issue  
// in your application, is a function of your  
// architecture as well as alignment...  
  
// elsewhere... UNSIGNED* volatile_data;
```



NppProtect::NppProtect

NppProtect::NppProtect();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
NppProtect* pProtect = new NppProtect;
```



NppProtect::Protect

inline

VOID

```
NppProtect::Protect();
```

Protects a data structure from access on another thread until the `UnProtect` member is called.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's protect service.
Post-condition	The data structure is protected from access until a subsequent call to the <code>UnProtect</code> member is called.

Parameters

None

Return Value

None

Example

```
// assume that NppProtect* pProtect exists.

pProtect->Protect();

//do something critical to protect here
```



NppProtect::SetProtectedBYTE_PTR

```

inline
VOID
NppProtect::SetProtectedBYTE_PTR
(
    BYTE_PTR& bp,
    BYTE_PTR newBp
);

```

Sets the BYTE_PTR value stored in "bp" with "newBp", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Does a protected set on the BYTE_PTR.
Post-condition	The BYTE_PTR is set.

Parameters

Parameter	Overview
bp	BYTE_PTR value.
newBp	new BYTE_PTR value.

Return Value

None

Example

```

// assume that NppProtect* pProtect exists.

// elsewhere... CHAR* volatile_data;

CHAR* new_value;

pProtect->SetProtectedBYTE_PTR( volatile_data, new_value );

```



NppProtect::SetProtectedCHAR

```
inline
VOID
NppProtect::SetProtectedCHAR
(
    CHAR& c,
    CHAR newC
);
```

Sets the CHAR value stored in "c" with "newC", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Does a protected set on the CHAR .
Post-condition	The CHAR is set.

Parameters

Parameter	Overview
c	CHAR value.
newC	new CHAR value.

Return Value

None

Example

```
// assume that NppProtect* pProtect exists.

// elsewhere... CHAR volatile_data;

CHAR new_value;
```



NppProtect::SetProtectedINT

VOID

NppProtect::SetProtectedINT

```
(
    INT& i,
    INT newI
);
```

Sets the INT value stored in "i" with "newI", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Does a protected set on the INT.
Post-condition	The INT is set.

Parameters

Parameter	Overview
i	INT value.
newI	new INT value.

Return Value

None

Example

```
// assume that NppProtect* pProtect exists.

// elsewhere... INT volatile_data;

INT new_value;

pProtect->SetProtectedINT( volatile_data, new_value );
```



NppProtect::SetProtectedUNSIGNED

```
inline
VOID
NppProtect::SetProtectedUNSIGNED
(
    UNSIGNED& u,
    UNSIGNED newU
);
```

Sets the UNSIGNED value stored in "u" with "newU", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Does a protected set on the UNSIGNED .
Post-condition	The UNSIGNED is set.

Parameters

Parameter	Overview
u	UNSIGNED value.
newU	new UNSIGNED value.

Return Value

None

Example

```
// assume that NppProtect* pProtect exists.

// elsewhere... UNSIGNED volatile_data;

UNSIGNED new_value;

pProtect->SetProtectedUNSIGNED( volatile_data, new_value );
```



NppProtect::SetProtectedUNSIGNED_PTR

```

inline
VOID
NppProtect::SetProtectedUNSIGNED_PTR
(
    UNSIGNED_PTR& up,
    UNSIGNED_PTR

```

Sets the UNSIGNED_PTR value stored in "up" with "newUp", protecting the variable from multiple thread access using this protection object.

Overview

Condition	Description
Pre-condition	None
Action	Does a protected set on the UNSIGNED_PTR.
Post-condition	The UNSIGNED_PTR is set.

Parameters

Parameter	Overview
up	UNSIGNED_PTR value.
newUp	new UNSIGNED_PTR value.

Return Value

None

Example

```

// assume that NppProtect* pProtect exists.

// elsewhere... UNSIGNED* pData;

UNSIGNED* pNewValue;

pProtect->SetProtectedUNSIGNED_PTR( pData, pNewValue );

```



NppProtect::UnProtect

inline

VOID

NppProtect::UnProtect();

Releases protection of the data structure to other threads.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's unprotect service.
Post-condition	The data structure is not protected from access by other threads.

Parameters

None

Return Value

None

Example

```
// assume that NppProtect* pProtect exists.

pProtect->Protect();

//do something critical to protect here

pProtect->UnProtect();
```



class PartitionPool : public NucleusPlus

A partition memory pool object contains a specific number of fixed-size memory partitions. The memory location of the pool, the number of bytes in the pool, and the number of bytes in each partition are determined by the application. If the memory location of the pool is not provided, a memory block of the appropriate size is allocated from the free store. Individual partitions are allocated and deallocated from the partition-memory pool.

Suspension

The `Allocate` member provides options for unconditional suspension, suspension with a time-out, and no suspension. A task attempting to allocate a partition from an empty pool can suspend. Resumption of that task is possible when a partition is returned to the pool. Multiple tasks may suspend on a single partition memory pool. Tasks are suspended in either FIFO or priority order, depending on how the partition memory pool was created. If the partition memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the partition memory pool supports priority suspension, tasks are resumed from high priority to low priority. All Nucleus C++ members that provide suspension options have their `suspend` parameter set to `NU_SUSPEND` by default. This means that if the parameter is absent (`or NU_SUSPEND`), tasks are suspended if the request cannot be satisfied.

Dynamic Creation

Nucleus C++ partition memory pool objects are created and deleted dynamically. There is no preset limit on the number of `PartitionPool` objects an application may have.

Determinism

Since searching is completely avoided, processing required for allocating and deallocating partitions is fast and constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the partition memory pool.



Partition Information

Application tasks may obtain a list of active partition memory pools. Detailed information about each partition memory pool is also available. This information includes the partition memory pool name, starting pool address, total partitions, partition size, remaining partitions, number of tasks suspended, and the identity of the first suspended tasks.

Public Member Functions

Member	Overview
<code>~PartitionPool</code>	Destructor
<code>Allocate</code>	Allocates a block of memory from this <code>PartitionPool</code> object.
<code>Deallocate</code>	Returns the block of memory to this <code>PartitionPool</code> object.
<code>PartitionPool</code>	Constructor
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the info object.



PartitionPool::~~PartitionPool

virtual

PartitionPool::~~PartitionPool();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete partition pool service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



PartitionPool::Allocate

```
inline
STATUS
PartitionPool::Allocate
(
    VOID* returnPointer,
    UNSIGNED suspend =
    NU_SUSPEND
);
```

Allocates a block of memory from this PartitionPool object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's allocate partition pool service.
Post-condition	If successful, the partition pool has one less partition.

Parameters

Parameter	Overview
returnPointer	Pointer to the caller's memory pointer.
suspend	Specifies whether or not to suspend the calling task if there are no memory partitions available.

Return Value

Return Value	Overview
aValidPointer	A pointer to the allocated memory block.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_POOL	Indicates the memory pool pointer is invalid.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_NO_PARTITION	Indicates the memory partition request could not be immediately satisfied.
NU_POOL_DELETED	Memory pool was deleted while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that no memory partition is available even after suspending for the specified timeout value.



Example

```
PartitionPool* commBufferPool =
new PartitionPool("cbuff",    // partition name
                  2048,      // total pool size
                  50,        // partition size
                  NU_PRIORITY // suspension ordering
                  );

void* pPartition;
STATUS status;

// allocate a buffer, specify no suspension
status = commBufferPool.Allocate(&pPartition,
                                NU_NO_SUSPEND);

if (status != NU_SUCCESS)
{
    // the allocation failed...
}
```



PartitionPool::Deallocate

```
inline
STATUS
PartitionPool::Deallocate
(
    VOID* partition
);
```

Returns the block of memory to this `PartitionPool` object.

Overview

Condition	Description
Pre-condition	The block of memory was allocated using the <code>Allocate</code> member of this object.
Action	Uses the kernel's deallocate partition pool service.
Post-condition	If successful, the partition pool has one more partition.

Parameters

Parameter	Overview
partition	Points to the partition of memory to return.

Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_SUCCESS	Indicates successful completion of the service.



Example

```
PartitionPool* commBufferPool =  
new PartitionPool("cbuff",    // partition name  
                  2048,      // total pool size  
                  50,        // partition size  
                  NU_PRIORITY // suspension ordering  
                  );  
  
void* pPartition;  
STATUS status;  
  
// allocate a buffer, specify no suspension  
status = commBufferPool.Allocate(&pPartition,  
                                 NU_NO_SUSPEND);  
  
if (status == NU_SUCCESS)  
{  
    // the allocation was good  
    // do something with the memory  
    commBufferPool.Deallocate( pPartition );  
}
```



PartitionPool::PartitionPool

PartitionPool::PartitionPool

```
(
const CHAR* name,
    UNSIGNED size,
    UNSIGNED
partitionSize,
    OPTION suspendType,
    VOID* startAddress = NULL
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's partition pool create service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The name of PartitionPool.
size	The size of Partition Pool.
partitionSize	The size of the partition.
suspendType	Specifies how tasks suspend on an object.
startAddress	An optional starting address for pool. A standard operating system allocation is performed and a starting address assigned if the parameter is absent or zero.

Return Value

None

Example

```
PartitionPool* commBufferPool =
new PartitionPool("cbuff",    // partition name
                2048,        // total pool size
                50,          // partition size
                NU_PRIORITY // suspension ordering
);
```



PartitionPool::UpdateInfo

```
inline
PartitionPoolInfo&
PartitionPool::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the info object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's partition pool information service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_POOL</code>	Indicates the partition pool pointer is invalid.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// get the latest info on our partition pool
// from the kernel

const PartitionPoolInfo& info =
commBufferPool.UpdateInfo();
```



class PartitionPoolInfo : public PoolInfo

Derived `PoolInfo` class that holds various information about `PartitionPool` objects.

Public Member Functions

Member	Overview
<code>~PartitionPoolInfo</code>	Destructor
<code>GetControlBlock</code>	Returns a const reference to the kernel's partition pool control block for the <code>PartitionPool</code> this <code>PartitionPoolInfo</code> object is associated with.
<code>GetPartitionsAllocated</code>	Returns the number of allocated pool partitions
<code>GetPartitionSize</code>	Returns the number of bytes in each partition.
<code>PartitionPoolInfo</code>	Constructor
<code>Update</code>	Updates the information object for the <code>PartitionPool</code> .



PartitionPoolInfo::~~PartitionPoolInfo

virtual

`PartitionPoolInfo::~~PartitionPoolInfo();`

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



PartitionPoolInfo::GetControlBlock

```
inline
NU_PARTITION_POOL&
PartitionPoolInfo::GetControlBlock()
const;
```

Returns a const reference to the kernel's partition pool control block for the PartitionPool this PartitionPoolInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidReference	A const reference to the kernel's memory pool control block for the PartitionPool this PartitionPoolInfo object is associated with.

Example

```
// assume PartitionPool commBufferPool exists.

// get the latest info from the kernel
const PartitionPoolInfo& info =
commBufferPool.UpdateInfo();
```



PartitionPoolInfo::GetPartitionsAllocated

```
inline
UNSIGNED
PartitionPoolInfo::GetPartitionsAllocated()
const;
```

Returns the number of allocated pool partitions.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of allocated pool partitions.

Example

```
// assume PartitionPool commBufferPool exists.

// get the latest info from the kernel
const PartitionPoolInfo& info =
commBufferPool.UpdateInfo();
```



PartitionPoolInfo::GetPartitionSize

```
inline  
UNSIGNED  
PartitionPoolInfo::GetPartitionSize()  
const;
```

Returns the number of bytes in each partition.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of bytes in each partition.

Example

```
// assume PartitionPool commBufferPool exists.  
  
// get the latest info from the kernel  
const PartitionPoolInfo& info =  
commBufferPool.UpdateInfo();  
  
cout << "Partition Size" << info.GetPartitionSize();
```



PartitionPoolInfo::PartitionPoolInfo

```
PartitionPoolInfo::PartitionPoolInfo();
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Constructor for PartitionPoolInfo objects.
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

PartitionPoolInfo::Update

virtual

STATUS

PartitionPoolInfo::Update();

Updates the information object for the PartitionPool.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's partition pool information service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_POOL	Indicates the partition pool pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that PartitionPool* pPool exists.  
  
// create and info object.  
PartitionPoolInfo info;  
  
// assign it information from the PartitionPool object  
info = pPool->UpdateInfo();
```



class Pipe : public NucleusPlus

Pipes provide a mechanism to transmit multiple messages. Messages are sent and received by value. A send-message request copies the message into the pipe, while receive-message request copies the message out of the pipe. Messages may be placed at the front of the pipe or at the back of the pipe.

Message Size

A message consists of one or more bytes. Both fixed and variable-length messages are supported. The type of message format is defined when the pipe is created. Variable length message pipes require an additional 32-bit word of overhead for each message in the pipe. Additionally, receive-message requests on variable-length message pipes specify the maximum message size, while the same request on fixed-length message pipes specify the exact message size.

Suspension

Send and receive pipe services provide options for unconditional suspension, suspension with a time-out, and no suspension. Task objects may suspend on a pipe for several reasons. Tasks attempting to receive a message from an empty pipe can suspend. Also, a task attempting to send a message to a full pipe can suspend. A suspended task is resumed when the pipe is able to satisfy that task's request. For example, suppose a task is suspended on a pipe waiting to receive a message. When a message is sent to the pipe, the suspended task is resumed. Multiple tasks may suspend on single pipe. Tasks are suspended in either FIFO or priority order, depending on how the pipe was created. If the pipe supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the pipe supports priority suspension, tasks are resumed from high priority to low priority. If a Nucleus C++ Pipe member has a suspension option, this option is set to `NU_SUSPEND` by default. This means that if the parameter is absent, tasks are suspended if the request cannot be satisfied.

Broadcast

A pipe message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the pipe are given the broadcast message.

Dynamic Creation

Nucleus C++ pipes are created and deleted dynamically. There is no preset limit on the number of Pipe objects an application may have.



Determinism

Basic processing time required for sending and receiving pipe messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the pipe.

Pipe Information

Application tasks may obtain a list of active pipes. Detailed information about each pipe can also be obtained. This information includes the pipe name, message format, suspension type, number of messages present, and the first task waiting.

Public Member Functions

Member	Overview
<code>~Pipe</code>	Destructor
<code>Broadcast</code>	Broadcasts a message to this <code>Pipe</code> object.
<code>Pipe</code>	Constructor
<code>Receive</code>	Retrieves a message from this <code>Pipe</code> object.
<code>Reset</code>	Discards all messages currently in this <code>Pipe</code> object.
<code>Send</code>	Sends a message to this <code>Pipe</code> object.
<code>SendToFront</code>	Places a message at the front of this <code>Pipe</code> object.

Public Member Functions

Member	Overview
<code>UpdateInfo</code>	Updates internal object information, and returns a reference to the info object.



Pipe::~Pipe

virtual

`Pipe::~Pipe();`

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete pipe service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Pipe::Broadcast

```

inline
STATUS
Pipe::Broadcast
(
    VOID* object,
    UNSIGNED size,
    UNSIGNED suspend =
    NU_SUSPEND
);

```

Broadcasts a message to this `Pipe` object. If there is enough space in the pipe to hold the message, the service is processed immediately. Broadcasting the message to all tasks waiting for a message from this `Pipe` object. If no tasks are waiting, the message is placed at the back of the pipe.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's broadcast pipe service.
Post-condition	If successful, the message has been broadcast.

Parameters

Parameter	Overview
<code>object</code>	A pointer to the message to broadcast.
<code>size</code>	Specifies the number of bytes in the message suspend. Specifies whether or not to suspend the calling task if the pipe is full. <code>NU_NO_SUSPEND</code> causes the service to return immediately regardless of whether the request can be satisfied. <code>NU_SUSPEND</code> causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_FULL	Indicates the pipe is full.
NU_PIPE_RESET	Pipe was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the pipe is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Pipe* mypipe exists.

char message[20];

// broadcast the 20 byte message. Use the default
// parameter for suspend (NU_SUSPEND)
mypipe->Broadcast( &message, 20 );
```



Pipe::Pipe

```
Pipe::Pipe
(
    CHAR* name,
    UNSIGNED numberBytes,
    OPTION type,
    UNSIGNED objectSize,

    OPTION suspend
);
```

Constructor.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create pipe service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The pipe's name.
numberBytes	The total number of bytes in the pipe.
type	Specifies the type of messages managed by the pipe.
objectSize	Specifies the exact size of each object if the pipe supports fixed sized objects. Otherwise, objectSize specifies the maximum object size.
suspend	Specifies how tasks suspend on the pipe.

Return Value

None



Example

```
// example 1: create a pipe that holds
// 100 messages that are fixed size, the
// messages are 1 byte in size, and tasks
// suspend in fifo order trying to add or
// take data from the pipe.

Pipe* pPipe = new Pipe("bytepipe",    // pipe name
                       100,           // pipe size
                       NU_FIXED_SIZE  // message type
                       sizeof( BYTE ), // message size
                       NU_FIFO);      // susp.order

// example 2: create a pipe that holds
// 20 messages that are variable size, the
// max message size is 24 bytes, and tasks
// suspend in priority order trying to add or
// take data from the pipe.

Pipe* pPipe = new Pipe("varpipe",     // pipe name
                       20,             // pipe size
                       NU_VARIABLE_SIZE // type
                       24,             // message size
                       NU_PRIORITY);   // susp.order
```



Pipe::Receive

```

inline
STATUS
Pipe::Receive
(
    VOID* destination,
    UNSIGNED size,
    UNSIGNED* actualSize,

    UNSIGNED suspend = NU_SUSPEND
);

```

Retrieves a message from this `Pipe` object. If the pipe contains one or more messages, the message in front is immediately removed from the pipe and copied into the destination location.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's receive pipe service.
Post-condition	If successful, the message is received and removed from the pipe.

Parameters

Parameter	Overview
destination	A pointer to the message destination. Note: the message destination must be capable of holding "size" <code>UNSIGNED</code> data elements.
size	Specifies the number of <code>UNSIGNED</code> data elements in the message. This number must correspond to the message size defined when the <code>Pipe</code> object was created.
actualSize	A pointer to a variable to hold the actual number of <code>UNSIGNED</code> data elements in the received message.
suspend	Specifies whether or not to suspend the calling task if the pipe is empty. <code>NU_NO_SUSPEND</code> causes the service to return immediately, regardless of whether the request can be satisfied. <code>NU_SUSPEND</code> causes the task to suspend until a message is received. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until a message is received, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that <code>suspend</code> attempted from a non-task thread.
NU_MESSAGE_OVERFLOW	Indicates that the message destination was not large enough to hold the message.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended
NU_PIPE_EMPTY	Indicates the pipe is empty.
NU_PIPE_RESET	Pipe was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the pipe is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Pipe* mypipe exists.

// example 1: We are receiving messages that are
// variable size from the pipe. We know that the
// max size is 50... and we want to suspend for
// up to 100 ticks

    STATUS    status;
    char      message[50];
    UNSIGNED  actual;

    status =
    mypipe->Receive( &message, 50, &actual,100);

    if (status == NU_SUCCESS)
    {
        // do whatever depending on message size
        // here...

// example 2: We are receiving messages that are
// fixed size from the pipe. the size of the
// message is 10... and we don't want to suspend.

        STATUS    status;
        char      message[10];
        UNSIGNED  actual;
        status =
        mypipe->Receive( &message, 10, &actual,NU_NO_SUSPEND);

        if (status == NU_SUCCESS)
        {
            // yes, you got a message
```



Pipe::Reset

```
inline  
STATUS  
Pipe::Reset();
```

Discards all messages currently in this `Pipe` object. All tasks suspended on the pipe are resumed with the appropriate reset status.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's reset pipe service.
Post-condition	If successful, all of the messages have been discarded.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_PIPE</code>	Indicates the pipe pointer is invalid.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```
// assume that Pipe commPipe exists.
char    message[50];
UNSIGNED actual;// this call resets the content of the pipe
// and resumes all suspended tasks.
status = commPipe.Reset();
mypipe->Receive( &message, 50, &actual,100);

if (status == NU_SUCCESS)
{
    // do whatever depending on message size
    // here...

// example 2: We are receiving messages that are
// fixed size from the pipe. the size of the
// message is 10... and we don't want to suspend.

    STATUS    status;
    char      message[10];
    UNSIGNED  actual;

status =
mypipe->Receive( &message, 10, &actual,NU_NO_SUSPEND);

if (status == NU_SUCCESS)
{
    // yes, you got a message
```



Pipe::Send

```

inline
STATUS
Pipe::Send
(
    VOID* object,
    UNSIGNED size,
    UNSIGNED suspend = NU_SUSPEND
)

```

Sends a message to this Pipe object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's send pipe service.
Post-condition	If successful, the message is at the back of the pipe.

Parameters

Parameter	Overview
object	A pointer to the message to send.
size	Specifies the number of bytes in the message.
suspend	Specifies whether or not to suspend the calling task if the pipe is full. NU_NO_SUSPEND causes the service to return immediately, regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_FULL	Indicates the pipe is full.
NU_PIPE_RESET	Pipe was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the pipe is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Pipe* mypipe exists.

// Send a message of length 4 bytes with no
// suspension.

STATUS status;
UNSIGNED message;

status =
mypipe->Send( &message, sizeof(UNSIGNED),
              NU_NO_SUSPEND);
if (status == NU_SUCCESS)
{
    // your message went out!
```



Pipe::SendToFront

```
inline
STATUS
Pipe::SendToFront
(
    VOID* object,
    UNSIGNED size,
    UNSIGNED suspend =
    NU_SUSPEND
);
```

Places a message at the front of this Pipe object. If there is enough space in the pipe to hold the message, the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's send to front pipe service.
Post-condition	If successful, the message is at the front of the pipe.

Parameters

Parameter	Overview
object	A pointer to the message to send.
size	Specifies the number of bytes in the message.
suspend	Specifies whether or not to suspend the calling task if the pipe is full. NU_NO_SUSPEND causes the service to return immediately regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_FULL	Indicates the pipe is full.
NU_PIPE_RESET	Pipe was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the pipe is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Pipe* mypipe exists.

// Send a message of length 4 bytes and
// suspend until it goes out the door.

STATUS    status;
UNSIGNED message=0x12345678;

status =
mypipe->SendToFront( &message, sizeof(UNSIGNED),
                    NU_SUSPEND);

if (status == NU_SUCCESS)
{
    // your message went out!
```



Pipe::UpdateInfo

```
inline
PipeInfo&
Pipe::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information pipe service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Pipe* mypipe exists.
// get the latest scoop on your pipe.

const PipeInfo& info = mypipe->UpdateInfo();
```



class PipeInfo : public CommunicationInfo

Derived `CommunicationInfo` class that holds addition information applicable to `Pipe` objects.

Public Member Functions

Member	Overview
<code>~PipeInfo</code>	Destructor
<code>GetControlBlock</code>	Returns a const reference to the kernel's pipe control block for the <code>Pipe</code> this <code>PipeInfo</code> object is associated with.
<code>PipeInfo</code>	Constructor
<code>Update</code>	Updates the information object for the <code>Pipe</code> .



PipeInfo::~PipeInfo

virtual

```
PipeInfo::~PipeInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



PipeInfo::GetControlBlock

```
inline
NU_PIPE&
PipeInfo::GetControlBlock()
const;
```

Returns a const reference to the kernel's pipe control block for the Pipe this PipeInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Description
aValidReference	A const reference to the kernel's pipe control block for the Pipe this PipeInfo object is associated with.

Example

```
// assume that Pipe* mypipe exists.
// get the latest scoop on your pipe.

const PipeInfo& info = mypipe->UpdateInfo();

const NU_PIPE& pcb = info.GetControlBlock();
```



PipeInfo::PipeInfo

PipeInfo::PipeInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that Pipe* mypipe exists.  
// get the latest scoop on your pipe.
```



PipeInfo::Update

virtual

STATUS

PipeInfo::Update();

Updates the information object for the Pipe.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information pipe service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Pipe* mypipe exists.

// create and info object.
PipeInfo info;

// assign it information from the Pipe object
info = mypipe->UpdateInfo();

...

// get the latest information from the kernel.
info.Update();
```



class PoolInfo : public SuspendInfo

Derived `SuspendInfo` class that holds various information about `Pool` objects.

Public Member Functions

Member	Overview
<code>~PoolInfo</code>	Destructor
<code>GetBytesAvailable</code>	Returns the number of bytes available in the pool.
<code>GetPoolAddress</code>	Returns the starting address of pool.
<code>GetPoolSize</code>	Returns the number of bytes in the pool.
<code>PoolInfo</code>	Constructor



PoolInfo::~~PoolInfo

virtual

PoolInfo::~~PoolInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



PoolInfo::GetBytesAvailable

```
inline  
UNSIGNED  
PoolInfo::GetBytesAvailable()  
const;
```

Returns the number of bytes available in the pool.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of bytes available in the pool.

Example

```
//assume that MemoryPool* pCommPool exists.  
  
const MemoryPoolInfo& info =  
    pCommPool->UpdateInfo();
```



PoolInfo::GetPoolAddress

```
inline
VOID*
PoolInfo::GetPoolAddress()
const;
```

Returns the starting address of pool.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The starting address of pool.

Example

```
//assume that MemoryPool* pCommPool exists.

const MemoryPoolInfo& info =
    pCommPool->UpdateInfo();

VOID* pAddress = info.GetPoolAddress();
```



PoolInfo::GetPoolSize

```
inline  
UNSIGNED  
PoolInfo::GetPoolSize()  
const;
```

Returns the number of bytes in the pool.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of bytes in the pool.

Example

```
//assume that MemoryPool* pCommPool exists.  
  
const MemoryPoolInfo& info =  
    pCommPool->UpdateInfo();  
  
UNSIGNED size = info.GetPoolSize();
```



PoolInfo::PoolInfo

```
PoolInfo::PoolInfo();
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

None

Example

```
// PoolInfo is not used by itself. It is a base
// class for shared data between class MemoryPool
// and class PartitionPool.

//assume that MemoryPool* pCommPool exists.

MemoryPoolInfo info;
```



class Q : public NucleusPlus

Queues provide a mechanism for transmitting multiple messages. Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue. Messages may be placed at the front of the queue or at the back of the queue.

Public Member Functions

Member	Overview
~Q	Destructor
Broadcast	Broadcasts a message to this Q object.
Q	Constructor
Receive	Retrieves a message from the queue.
Reset	Resets this Q object.
Send	Places a message at the back of this Q object.
SendToFront	Places a message at the front of this Q object.

Public Member Functions

Member	Overview
UpdateInfo	Updates internal object information, and returns a reference to the info object.



Q::~~Q

virtual

Q::~~Q();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete queue service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Q::Broadcast

STATUS

Q::Broadcast

```
(
    VOID* object,
    UNSIGNED size,
    UNSIGNED suspend = NU_SUSPEND
);
```

Broadcasts a message to this Q object. Broadcasts the message to all tasks waiting for a message from this queue. If no tasks are waiting, the message is placed at the back of the queue. If there is enough space in the queue to hold the message, the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's broadcast queue service.
Post-condition	If successful, the message is broadcast.

Parameters

Parameter	Overview
object	Pointer to the message to broadcast.
size	Specifies the number of UNSIGNED data elements in the message.
suspend	Specifies whether or not to suspend the calling task if the queue is full. NU_NO_SUSPEND causes the service to return immediately regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_FULL	Indicates the queue is full.
NU_QUEUE_RESET	Queue was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Q* myQ exists.

char message[20];

// broadcast the 20 byte message. Suspend for up to
// 100 timer ticks.
myQ->Broadcast( &message, 20, 100 );
```



Q::Q

Q::Q

```
(
const CHAR* name,
    UNSIGNED numberUnsigned,
    OPTION type,
    UNSIGNED objectSize,

OPTION suspend
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create queue service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The queue's name.
numberUnsigned	The total number of UNSIGNED data elements in the queue.
type	Specifies the type of objects managed by the queue.
objectSize	Specifies the exact size of each object, if the queue supports fixed sized objects. Otherwise, objectSize specifies the maximum object size.
suspend	Specifies how tasks suspend on the queue.

Return Value

None



Example

```

// example 1: create a queue that holds
// 100 messages that are fixed size, the
// messages are 8 bytes in size. With
// queues you specify the message size in terms
// of number of UNSIGNED elements in the message.
// so for a message that is an array of 2 UNSIGNED
// elements, we give a size of 2. and tasks
// suspend in fifo order trying to add or
// take data from the queue.

Q* queue = new Q("8byteQ",          // name
                100,                // number messages
                NU_FIXED_SIZE       // message type
                2,                  // message size
                NU_FIFO);           // susp.order

// example 2: create a queue that holds
// 100 messages that are variable size, with
// a maximum size of 10 UNSIGNED elements.
// Tasks suspend in priority order trying to add or
// take data from the queue.

Q* queue = new Q("varQ",            // name
                100,                // number messages
                NU_VARIABLE_SIZE    // message type
                10,                  // max message size
                NU_PRIORITY);       // susp.order

```



Q::Receive

```
inline
STATUS
Q::Receive
(
    VOID* destination,
    UNSIGNED size,
    UNSIGNED* actualSize,
    UNSIGNED suspend = NU_SUSPEND
);
```

Retrieves a message from the queue. If the queue contains one or more messages, the message in front is immediately removed from the queue and copied into the destination location.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's receive queue service.
Post-condition	If successful, the message is received and removed from the queue.

Parameters

Parameter	Overview
destination	A pointer to the message destination. Note: the message destination must be capable of holding "size"UNSIGNED data elements.
size	Specifies the number of UNSIGNED data elements in the message.
actualSize	This number actualSize is a pointer to a variable to hold the actual number of UNSIGNED data elements in the received message.
suspend	Specifies whether or not to suspend the calling task if the queue is empty. NU_NO_SUSPEND causes the service to return immediately, regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until a message is sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until a message is sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_EMPTY	Indicates the queue is empty.
NU_QUEUE_RESET	Queue was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.



Example

```
// assume that Q* myQ exists.

// example 1: We are receiving messages that are
// variable size from the Q. We know that the
// max size is 50... and we want to suspend for
// up to 100 ticks
    STATUS    status;
    char      message[50];
    UNSIGNED  actual;

    status =
myQ->Receive( &message, 50, &actual,100);
    if (status == NU_SUCCESS)
    {
        // do whatever depending on message size
        // here...
// example 2: We are receiving messages that are
// fixed size from the queue. we know that the
// size is 10, and we do not want to suspend
// waiting for the message.
        STATUS    status;
        char      message[10];
        UNSIGNED  actual;

        status =
myQ->Receive( &message, 10, &actual,NU_NO_SUSPEND);
    if (status == NU_SUCCESS)
    {
        // yes, you got a message
```



Q::Reset

```
inline
STATUS
Q::Reset();
```

Resets this `Q` object. Discards all objects currently in this `Q` object. All `Task` objects suspended on the queue are resumed with the appropriate reset status.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's reset queue service.
Post-condition	If successful, the queue is empty.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_POINTER</code>	Indicates the message pointer is <code>NULL</code> .
<code>NU_INVALID_QUEUE</code>	Indicates the queue pointer is invalid.
<code>NU_INVALID_SIZE</code>	Indicates an invalid <code>size</code> request.
<code>NU_INVALID_SUSPEND</code>	Indicates that <code>suspend</code> attempted from a non-task thread.
<code>NU_QUEUE_DELETED</code>	Queue was deleted while the task was suspended.
<code>NU_QUEUE_FULL</code>	Indicates the queue is full.
<code>NU_QUEUE_RESET</code>	Queue was reset while the task was suspended.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_TIMEOUT</code>	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Q* myQ exists.
```



Q::Send

```
inline
STATUS
Q::Send
(
    VOID* object,
    UNSIGNED size ,
    UNSIGNED suspend = NU_SUSPEND
```

Places a message at the back of this Q object. If there is enough space in the queue to hold the message, the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's send queue service.
Post-condition	If successful, the message at the back of the queue.

Parameters

Parameter	Overview
object	A pointer to the message to send.
size	Specifies the number of UNSIGNED data elements in the message.
suspend	Specifies whether or not to suspend the calling task if the queue is full. NU_NO_SUSPEND causes the service to return immediately, regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that <code>suspend</code> attempted from a non-task thread.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_FULL	Indicates the queue is full.
NU_QUEUE_RESET	Queue was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Q* myQ exists.

// Send a message of length equal to 1 UNSIGNED
// elements, with no suspension.

STATUS    status;
UNSIGNED message;

status =
myQ->Send( &message, 1, NU_NO_SUSPEND);

if (status == NU_SUCCESS)
{
    // your message went out!
```



Q::SendToFront

```
inline
STATUS
Q::SendToFront
(
    VOID* object,
    UNSIGNED size ,
    UNSIGNED suspend =
    NU_SUSPEND
);
```

Places a message at the front of this Q object. If there is enough space in the queue to hold the message, the service is processed immediately.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's send to front service.
Post-condition	If successful, the message is at the front of the queue.

Parameters

Parameter	Overview
object	A pointer to the message to send.
size	Specifies the number of UNSIGNED data elements in the message.
suspend	Specifies whether or not to suspend the calling task if the queue is full. NU_NO_SUSPEND causes the service to return immediately, regardless of whether the request can be satisfied. NU_SUSPEND causes the task to suspend until the message can be sent. This value can be a timeout value between 1 and 4,294,967,293. The calling task will be suspended until the message can be sent, or timeout number of timer ticks have expired.



Return Value

Return Value	Overview
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_SIZE	Indicates an invalid <code>size</code> request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_FULL	Indicates the queue is full.
NU_QUEUE_RESET	Queue was reset while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.

Example

```
// assume that Q* myQ exists.

// Send a message of length equal to 1 UNSIGNED
// elements, with no suspension.

STATUS    status;
UNSIGNED message;

status =
myQ->SendToFront( &message, 1, NU_NO_SUSPEND);

if (status == NU_SUCCESS)
{
    // your message went out!
```



Q::UpdateInfo

```
inline  
QInfo&  
Q::UpdateInfo() const;  
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information queue service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Q* myQ exists.  
  
// have the kernel give us the latest on the Q.  
const QInfo& info = myQ->UpdateInfo();
```



class QInfo : public CommunicationInfo

Holds various information about Q objects.

Public Member Functions

Member	Overview
~Qinfo	Destructor
GetControlBlock	Returns a const reference to the kernel's queue control block for the Q object this QInfo object is associated with.
Qinfo	Constructor
Update	Updates information on the Q object.



QInfo::~~QInfo

virtual

```
QInfo::~~QInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



QInfo::GetControlBlock

```
inline
NU_QUEUE&
QInfo::GetControlBlock()
const;
```

Returns a const reference to the kernel's queue control block for the Q object this QInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidReference	A const reference to the kernel's queue control block for the Q this QInfo object is associated with.

Example

```
// assume that Q* myQ exists.

const QInfo& info = myQ->UpdateInfo();

const NU_QUEUE& qcb = info.GetControlBlock();
```



QInfo::QInfo

QInfo::QInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that Q* myQ exists.  
  
QInfo info = myQ->UpdateInfo();
```



QInfo::Update

virtual

STATUS

QInfo::Update();

Updates information on the Q object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information queue service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Q* queue exists.

QInfo info;    // create an info object.

// assign it information from the Q object
info = queue->UpdateInfo();

...

// get the latest information from the kernel.
info.Update();
```



class Semaphore : public NucleusPlus

The Nucleus C++ PLUS classes Semaphore, PrioritySemaphore, and FifoSemaphore are used to manage multithreaded access to critical shared resources. These critical section management classes are implemented using Nucleus PLUS semaphore services.

Public Member Functions

Member	Overview
~Semaphore	Destructor
Obtain	Obtain semaphore request.
Release	Release semaphore request.
Reset	Resets this Semaphore object to the value of initialCount.
Semaphore	Constructor
UpdateInfo	Updates internal object information, and returns a reference to the info object.



Semaphore::~Semaphore

virtual

Semaphore::~Semaphore();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete semaphore service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Semaphore::Obtain

```
virtual
inline
STATUS
Semaphore::Obtain
(
    UNSIGNED suspend = NU_SUSPEND
);
```

Obtain semaphore request. Since instances are created with an internal counter, obtaining a `Semaphore` object translates into decrementing the semaphore's internal counter by one. If the semaphore counter is zero before this call, the service cannot be immediately satisfied.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's obtain semaphore service.
Post-condition	If successful, the semaphore counter is decremented.

Parameters

Parameter	Overview
suspend	Specifies whether or not to suspend the calling task if the semaphore cannot be obtained. If <code>suspend</code> is set to <code>NU_NO_SUSPEND</code> , the service returns immediately regardless of whether or not the request can be satisfied. Otherwise, <code>suspend</code> contains a timeout value between 1 and 4,294,967,293. The calling task is suspended until the semaphore can be obtained or until the specified number of timer ticks (timeout value) have expired. The default value for <code>suspend</code> is <code>NU_SUSPEND</code> .



Return Value

Return Value	Overview
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_SEMAPHORE_DELETED	The semaphore was deleted while the task was suspended.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	Indicates that the semaphore is still unavailable even after suspending for the specified timeout value.
NU_UNAVAILABLE	Indicates the semaphore is unavailable.

Example

```
// Assume that Semaphore commSemaphore exists.
// Obtain the semaphore and suspend until we
// get it
commSemaphore.Obtain();
    // do your critical stuff here
commSemaphore.Release();

// example 2:
// Obtain the semaphore and suspend for
// up to 1500 milliseconds to get it
STATUS status;

status = commSemaphore.Obtain(TicksFromMS(1500));

if (status == NU_SUCCESS)
{
    // you got it, congratulations...

commSemaphore.Release();
```



Semaphore::Release

virtual

inline

STATUS

Semaphore::Release();

Releases semaphore request. If there are any tasks waiting to obtain the same semaphore, the first task waiting is given this instance of the semaphore. Otherwise, if there are no tasks waiting for this semaphore, the internal semaphore counter is incremented by one.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's release semaphore service.
Post-condition	If successful, the semaphore counter is incremented.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// Assume that Semaphore commSemaphore exists.
// Obtain the semaphore and suspend until we
// get it
commSemaphore.Obtain();
    // do your critical stuff here
commSemaphore.Release();
// example 2:
// Obtain the semaphore and suspend for
// up to 1500 milliseconds to get it
STATUS status;
status = commSemaphore.Obtain(TicksFromMS(1500));
if (status == NU_SUCCESS)
{
    // you got it, congratulations...
    // do your thing, and then release it.
    commSemaphore.Release();
}
```



Semaphore::Reset

```
virtual
inline
STATUS
Semaphore::Reset
(
    UNSIGNED initialCount = 1
);
```

Resets this Semaphore object to the value of `initialCount`. All tasks suspended on the semaphore are resumed with the appropriate reset status.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's reset semaphore service.
Post-condition	If successful, the semaphore counter is set to the supplied input parameter.

Parameters

Parameter	Overview
<code>initialCount</code>	The initial semaphore count, default to 1.

Return Value

Return Value	Overview
<code>NU_INVALID_SEMAPHORE</code>	Indicates the semaphore pointer is invalid.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// Assume that Semaphore commSemaphore exists.

commSemaphore.Reset();
```



Semaphore::Semaphore

Semaphore::Semaphore

```
(
const CHAR* name,
    UNSIGNED initialCount,
    OPTION suspend
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create semaphore service.
Post-condition	The object exists. The semaphore counter is set to the supplied input parameter.

Parameters

Parameter	Overview
name	The name of the semaphore.
initialCount	Specifies the initial count of the semaphore.
suspend	Specifies how tasks suspend on the semaphore. If suspend is NU_FIFO, the tasks suspend first-in-first-out. If suspend is NU_PRIORITY, the tasks suspend in priority-order.

Return Value

None

Example

```
// Create a semaphore named "ed" with an initial
// count of 1, that suspends callers in priority
// order
Semaphore* ed = new Semaphore("ed",1,NU_PRIORITY);
// Create a semaphore named "herm" with an initial
// count of 10, that suspends callers in FIFO
// order
Semaphore* herm = new Semaphore("herm",10,NU_FIFO);
```



Semaphore::UpdateInfo

```
inline
SemaphoreInfo&
Semaphore::UpdateInfo() const;
;
```

Updates internal object information, and returns a reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information semaphore service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// get the latest status on the semaphore. Inquiring
// minds want to know!
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();
```



class SemaphoreInfo : public SuspendInfo

The Nucleus C++ PLUS classes Semaphore, PrioritySemaphore, and FifoSemaphore are used to manage multithreaded access to critical shared resources. These critical section management classes are implemented using Nucleus PLUS semaphore services. Derived SuspendInfo class that holds various information about Semaphore objects.

Public Member Functions

Member	Overview
~SemaphoreInfo	Destructor
GetControlBlock	Returns a const reference to the kernel's queue control block for the semaphore this SemaphoreInfo object is associated with.
GetCount	Returns the current internal counter in the semaphore object.
SemaphoreInfo	Constructor
Update	Updates internal object information, and returns a reference to the info object.



SemaphoreInfo::~~SemaphoreInfo

virtual

SemaphoreInfo::~~SemaphoreInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



SemaphoreInfo::GetControlBlock

```
inline
NU_SEMAPHORE&
SemaphoreInfo::GetControlBlock()
const;
```

Returns a `const` reference to the kernel's queue control block for the Semaphore this `SemaphoreInfo` object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	A <code>const</code> reference to the kernel's queue control block for the semaphore this <code>SemaphoreInfo</code> object is associated with.

Example

```
// get the latest status on the semaphore.
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();

// get a reference to the control block.
const NU_SEMAPHORE& scb = info.GetControlBlock();
```



SemaphoreInfo::GetCount

```
inline
UNSIGNED
SemaphoreInfo::GetCount()
const;
```

Returns the current internal counter in the Semaphore object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current internal counter in the semaphore object.

Example

```
// get the latest status on the semaphore.
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();

UNSIGNED count = info.GetCount();
```



SemaphoreInfo::SemaphoreInfo

SemaphoreInfo::SemaphoreInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// get the latest status on the semaphore.  
SemaphoreInfo info = commSemaphore.UpdateInfo();
```



SemaphoreInfo::Update

virtual

STATUS

SemaphoreInfo::Update();

Updates internal object information.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information semaphore service.
Post-condition	The data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Semaphore* sem exists.

SemaphoreInfo info;    // create an info object.

// assign it information from the Semaphore object
info = sem->UpdateInfo();

...
```



class `Signal`

The Nucleus C++ PLUS class `Signal` is a class that manages asynchronous software interrupts in an embedded application using the Nucleus PLUS signaling services. This is a synchronization mechanism used for inter-task communication. The class `Task` has special software to support this advanced mechanism when enabled.

Public Member Functions

Member	Overview
<code>~Signal</code>	Destructor
<code>Disable</code>	Disables this <code>Signal</code> object.
<code>Enable</code>	Enables this <code>Signal</code> object.
<code>Send</code>	Sends this <code>Signal</code> object to the task.
<code>Signal</code>	Constructor
<code>Signal</code>	Creates signal for specifically for the supplied <code>Task t</code> .
<code>SignalHandler</code>	Actual signal handler for this <code>Signal</code> that is executed on its associated <code>Task</code> object's context (asynchronously).



Signal::~Signal

virtual

Signal::~Signal();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Releases the signal from the task.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Signal::Disable

```
inline  
VOID  
Signal::Disable();
```

Disables this Signal object.

Overview

Condition	Description
Pre-condition	None
Action	Coordinates the disabling of Signal with the Task we are signaling.
Post-condition	Signal is disabled.

Parameters

None

Return Value

None

Example

```
// Assume that SleepSignal* pSignal exists.  
  
pSignal->Disable();  
  
// see Signal class for a thorough example
```



Signal::Enable

VOID

`Signal::Enable();`

Enables this Signal object.

Overview

Condition	Description
Pre-condition	None
Action	Coordinates the enabling of this Signal with the Task we are signaling.
Post-condition	Signal is enabled.

Parameters

None

Return Value

None

Example

```
// Assume that SleepSignal* pSignal exists.

pSignal->Enable();

// see Signal::Signal for a thorough example.
```



Signal::Send

```
inline  
STATUS  
Signal::Send();
```

Sends this Signal object to the task.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's send signal service.
Post-condition	If successful, the signal was sent to the associated task.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

See example for `Signal::Signal(Task*)`.



Signal::Signal

Signal::Signal

```
(
    Task* t
);
```

Creates signal specifically for the supplied Task `t`.

Overview

Condition	Description
Pre-condition	None
Action	Registers this signal object with Task <code>t</code> .
Post-condition	The object exists. This signal object is registered with Task <code>t</code> but disabled.

Parameters

Parameter	Overview
<code>t</code>	The task to signal with this object.

Return Value

None



Example

```
// This code is not complete, but everything that is relevant to the
// discussion is here.

// system initialization
NppCreateMultitasking( void* first_available_memory )
{
    // component initialization here
    ...

    // create the ChildTask, the one we want
    // to be able to put to sleep on command.

    ChildTask* pChildTask = new ChildTask;

    // create the "babysitter", the signal
    // that we will use to put the child
    // task to sleep.

    SleepSignal* pChildBabysitter
        = new SleepSignal( pChildTask );

    // create the parent task with a pointer
    // to the babysitter. The parentTask will
    // decide when the child sleeps and then use
    // the "babysitter" to make it happen.

    ParentTask* pParentTask
        = new ParentTask( pChildBabysitter );

    // Start both of the tasks.
    pParentTask->Start();
    pChildTask->Start();
}

// code for the child task.
void
ChildTask::Entry()
{
    // enable the signal that puts us to sleep.
    // this way the child has control over
    // when she can be put to sleep. May result
    // in anarchy in a family, but good in a real-time
    // system.
```



```

// One limitation of Task::Sleep is that it is a
// static function, and puts whatever the calling
// thread is to sleep. That means that if you
// make a call from task1 like task2->Sleep( 10 );
// it is task1 that sleeps, not task2! You could
// come up with a messaging scheme whereby task1
// sends task2 a message that when handled
// means "go to sleep", but what if you wanted to
// do it asynchronously? Signals provide an
// asynchronous way of making something happen
// on another threads context. Lets develop a
// derived signal that makes another thread go
// to sleep.

class SleepSignal : public Signal
{
public:
    SleepSignal( Task* pTask )
        : Signal( pTask ),
          sleepTicks( 0 )
    {
    }

    virtual ~SleepSignal() {}

    // use this member to signal the
    // other task to sleep
    void Sleep( UNSIGNED ticks )
    {
        sleepTicks = ticks;
        // activate the signal. The next time
        // the task is scheduled, the signal
        // handler will run and the task will
        // sleep...
        Send();
    }
protected:

    UNSIGNED sleepTicks;

    // derived signal handler
    virtual
    void
    SignalHandler()
    {
        // this is executed on the receiving
        // tasks context.
        Task::Sleep( sleepTicks );
    }
}

```



```
while( 1 )
{
    pChildBabysitter->Enable();
    ...
    pChildBabysitter->Disable();
}

// elsewhere in your code for the ParentTask
void
ParentTask::Entry()
{
    // here we want to put the child to sleep
    // for 20000 milliseconds, the average sleeping
    // period for a 6 month old.

    pChildBabysitter->Sleep( TicksFromMS(20000) );

    // as soon as the Child is next scheduled, it
    // will go to sleep!

    // usage note: The SleepSignal class
    // is not thread safe for more than one
    // thread to use the same instance. This
    // is because we set a data member in one
    // thread that is used in another thread,
    // making the assumption that it will not
    // be changed from the time that it is set
    // to the time that it is used. This will
    // be true if only one thread is in control.
    // If you had multiple threads that wanted
    // to make the child sleep, they could each
    // have their own instance of the
    // SleepSignal...
```



Signal::Signal

Constructor. By default, the `Signal` is created for the currently executing `Task`.

Overview

Condition	Description
Pre-condition	None
Action	Registers this signal object with the current executing task.
Post-condition	The object exists. This signal object is registered with the calling task but disabled.

Parameters

None

Return Value

None

Example

```
// This constructor creates a signal for the
// current thread.

class MySignal : public Signal
{
public:
    MySignal(); // default constructor
    ...

// an entry function for a derived task that is
// going to create a signal.
void
MyTask::Entry()
{
    // create the signal on the stack. The signal
    // will then execute on an instance of MyTask.
    MySignal signal;
    ...
}
```



Signal::SignalHandler

virtual

VOID

```
Signal::SignalHandler()=0;
```

Actual signal handler for this `Signal` that is executed on its associated `Task` object's context (asynchronously). Derived `Signal` objects will provide their own signal handling routine. For the base class `Signal`, this member is a pure virtual function.

Overview

Condition	Description
Pre-condition	Derived class specific behavior.
Action	Derived class specific behavior.
Post-condition	Derived class specific behavior.

Parameters

None

Return Value

None

Example

This member is called for you by the framework.

See example for `Signal::Signal(Task*)` for how to define your own derived signal class.



class SuspendInfo : public NucleusPlusInfo

Derived `NucleusPlusInfo` class that holds various information about Nucleus C++ objects that cause task suspension. Since this class contains a pure virtual function, objects of this class cannot be declared.

Public Member Functions

Member	Overview
<code>~SuspendInfo</code>	Destructor
<code>GetFirstTaskWaiting</code>	Returns the first task suspended on the object.
<code>GetNumberTasksWaiting</code>	Returns the number of tasks waiting on object.
<code>GetSuspendType</code>	Returns the task suspend type. This is how tasks using the object that can cause suspension suspend when necessary. Valid options are <code>Fifo</code> and <code>Priority</code> suspension.
<code>SuspendInfo</code>	Constructor



SuspendInfo::~~SuspendInfo

virtual

```
SuspendInfo::~~SuspendInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



SuspendInfo::GetFirstTaskWaiting

```
inline
Task*
SuspendInfo::GetFirstTaskWaiting()
const;
```

Returns the first task suspended on the object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A valid pointer to the first task suspended on the object.

Example

```
// SemaphoreInfo is a concrete class derived from
// SuspendInfo which is not...

// get the latest status on the semaphore.
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();

Task* pTask = info.GetFirstTaskWaiting();
```



SuspendInfo::GetNumberTasksWaiting

```
inline
UNSIGNED
SuspendInfo::GetNumberTasksWaiting()
const;
```

Returns the number of tasks waiting on the object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of tasks waiting on the object

Example

```
// SemaphoreInfo is a concrete class derived from
// SuspendInfo which is not...

// get the latest status on the semaphore.
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();

UNSIGNED impatientlyWaiting =
    info.GetNumberTasksWaiting();
```



SuspendInfo::GetSuspendType

```
inline
OPTION
SuspendInfo::GetSuspendType()
const;
```

Returns the task suspend type. This is how tasks using the object that can cause suspension suspend when necessary. Valid options are `Fifo` and `Priority` suspension.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	Valid options are <code>Fifo</code> and <code>Priority</code> suspension.

Example

```
// SemaphoreInfo is a concrete class derived from
// SuspendInfo which is not...

// get the latest status on the semaphore.
const SemaphoreInfo& info =
    commSemaphore.UpdateInfo();

OPTION suspend = info.GetSuspendType();

// this will either be NU_FIFO, or NU_PRIORITY
```



SuspendInfo::SuspendInfo

SuspendInfo::SuspendInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

SuspendInfo is not a concrete class, but is a base class for all other info objects that have Task suspension information. As such, you won't ever be creating one.



class SystemClock

A `SystemClock` object is a class that controls the system clock. There are no data members and only static routines. This object can be used without declaring any objects of this class since there is only one clock in the system.

Public Member Functions

Member	Overview
<code>~SystemClock</code>	Destructor
<code>SystemClock</code>	Constructor

Public Class Member Functions

Member	Overview
<code>Get</code>	This is a synonym for the <code>Retrieve</code> method. Returns the current value of the continuously incrementing timer tick counter. The timer ticks once for every timer interrupt.
<code>Retrieve</code>	Returns the current value of timertick counter.
<code>Set</code>	Sets the continuously counting system clock to the value specified.



SystemClock::~~SystemClock

virtual

```
SystemClock::~~SystemClock();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



SystemClock::Get

```
static
inline
UNSIGNED
SystemClock::Get();
```

This is a synonym for the `Retrieve` method. Returns the current value of the continuously incrementing `timertick` counter. The timer ticks once for every timer interrupt.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's retrieve system clock service.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>validData</code>	The current value of the continuously incrementing <code>timertick</code> counter.

Example

```
UNSIGNED currentClock = SystemClock::Get();
```



SystemClock::Retrieve

```
static  
inline  
UNSIGNED  
SystemClock::Retrieve();
```

Returns the current value of the continuously incrementing timertick counter. The timer ticks once for every timer interrupt.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's retrieve system clock service.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current value of the continuously incrementing timer tick counter.

Example

```
UNSIGNED currentClock = SystemClock::Retrieve();
```



SystemClock::Set

```

static
inline
VOID
SystemClock::Set
(
    UNSIGNED newValue
);

```

Sets the continuously counting system clock to the value specified.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's set system clock service.
Post-condition	The clock has been set.

Parameters

Parameter	Overview
newValue	The new value of the system clock.

Return Value

None

Example

```

// reset the system clock
SystemClock::Set( 0 );

```



SystemClock::SystemClock

SystemClock::SystemClock();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

The `SystemClock` object is a singleton object that is accessed only through its static functions. As such, you will never create one.



class Task : public NucleusPlus

The Nucleus C++ PLUS class `Task` is a base class used to create and manage multiple threads in an embedded application. The class is implemented using Nucleus PLUS real-time task services. Programmers either create new thread behaviors or reuse existing behaviors that solve related tasks.

Public Member Functions

Member	Overview
<code>~Task</code>	Destructor
<code>ChangePriority</code>	Changes priority of this <code>Task</code> object.
<code>ChangeTimeSlice</code>	Changes the time slice of this <code>Task</code> object.
<code>IsCurrent</code>	Returns TRUE if this is the current executing <code>Task</code> object.
<code>Reset</code>	Resets this previously terminated or finished <code>Task</code> object.
<code>Resume</code>	Resumes this previously suspended <code>Task</code> object.
<code>Start</code>	Starts this <code>Task</code> object.
<code>Suspend</code>	Unconditionally suspends this <code>Task</code> object.
<code>Task</code>	Constructor.
<code>Terminate</code>	Terminates this <code>Task</code> object.
<code>UpdateInfo</code>	Updates internal object information, and returns a <code>const</code> reference to the <code>info</code> object.

Protected Member Functions

Member	Overview
<code>Entry</code>	The <code>Task</code> object virtual entry point called by the kernel when the task is first scheduled.

Public Class Member Functions

Member	Overview
<code>CheckStack</code>	Checks the stack of current executing task object.
<code>Current</code>	Returns a pointer to the current executing <code>Task</code> object.
<code>GetTask</code>	Returns the pointer to the <code>Task</code> object associated with <code>NU_TASK</code> control block.
<code>PreemptionOff</code>	Turns preemption off for the currently executing task.
<code>PreemptionOn</code>	Turns preemption on for the currently executing task.
<code>Relinquish</code>	Allows all other ready tasks of the same priority a chance to execute before the currently executing task runs again.
<code>Sleep</code>	Suspends current executing task for specified timer ticks.



Task::~Task

virtual

Task::~Task();

Destructor. If this is a self-destructing task, we must obtain help destructing the object. This is because as soon as we terminate the task, the thread stops. It will not execute anymore! Therefore, we would never get to the code below the termination call that deallocates memory, deletes the RTOS task, and potentially deletes the task object, derived objects, members, etc. Also, since the thread is running, we cannot deallocate the memory associated with the stack and task object. This is because accessing after the deallocation could result in protection faults if the task is running on a processor and system that supports memory management, or if another task allocates the memory associated with the stack. This destructor calls `Terminate` to terminate the task. This coordinates the termination with the task, and provides the task with a chance to elegantly terminate.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete task service.
Post-condition	Upon exit, the task object is completely destructed. If this is a self-destruction call, this routine will actually never exit. It will <code>Suspend</code> until the task destructing helper task actually calls this routine to destruct the task.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Task::ChangePriority

```
inline
OPTION
Task::ChangePriority
(
    OPTION newPriority
);
```

Changes priority of this Task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's change priority task service.
Post-condition	The priority of the object has been changed.

Parameters

Parameter	Overview
newPriority	The new priority to change to.

Return Value

Return Value	Overview
validData	The previous priority of the Task.

Example

```
OPTION old_priority;

// change the priority to be 2
old_priority = mytask.ChangePriority( 2 );
```



Task::ChangeTimeSlice

```
inline
UNSIGNED
Task::ChangeTimeSlice
(
    UNSIGNED newTimeSlice
);
```

Changes the time slice of this Task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's change time slice task service.
Post-condition	The time slice of the object has been changed.

Parameters

Parameter	Overview
newTimeSlice	newTimeSlice is the new time slice specified. If newTimeSlice is 0, time slicing for this task is disabled.

Return Value

Return Value	Overview
validData	The previous time slice value.

Example

```
UNSIGNED old_slice;

// disable timeslicing for this task
old_slice = pTask->ChangeTimeSlice( 0 );

// restore it to the previous value
```



Task::CheckStack

```
static
inline
UNSIGNED
Task::CheckStack();
```

Checks the stack of current executing task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's check stack task service.
Post-condition	If the remaining amount of stack space is less than that required to save the caller's context, a stack overflow condition is present, and control will not return to the caller.

Parameters

None

Return Value

Return Value	Overview
validData	The number of free bytes remaining on the stack.

Example

```
void MyTask::Entry()
{
    UNSIGNED bytes_left;
```



Task::Current

```
static
inline
Task*
Task::Current();
```

Returns a pointer to the current executing Task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's current task pointer service and extracts the this pointer from the resulting task control block.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A pointer to the current executing Task object.

Example

```
Task* pTask;

// get a pointer to the currently executing task.
pTask = Task::Current();

// ( ... )
```



Task::Entry

virtual

VOID

Task::Entry()=0;

This is the actual Task object's virtual entry point that is called by the kernel when the task is first scheduled. For the base class Task, this member is a pure virtual function. Derived Task objects will define an Entry member specific to the requirements of the specific task.

Overview

Condition	Description
Pre-condition	Derived class specific behavior.
Action	Derived class specific behavior.
Post-condition	The task is put in the finished state and other derived class specific behavior.

Parameters

None

Return Value

None

Example

```
// This routine is called by the framework and doesn't need to be
// explicitly called. The exception is when you are extending one of
// your derived Task objects and you would like to call your base
// classes entry routine.

class MyTask : public Task
{
protected:
    virtual VOID Entry();...
};

class MyDerivedTask : public MyTask
{
protected:
    virtual VOID Entry();...
};

...
VOID
MyDerivedTask::Entry()
{
    // do the special processing necessary for
    // MyDerivedTask here
    // call the base classes entry routine
    MyTask::Entry();
}
```



Task::GetTask

```
static
inline
Task*
Task::GetTask
(
const NU_TASK* pTCB
);
```

Returns the pointer to the Task object associated with NU_TASK control block.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's current task pointer service.
Post-condition	None

Parameters

Parameter	Overview
pTCB	pTCB points to a task control block.

Return Value

Return Value	Overview
aValidPointer	A valid pointer to the task associated with NU_TASK control block.

Example

```
NU_TASK* pTCB;

// you have a pointer to a Task Control Block that
// you got from somewhere, maybe a C API call ...

Task* pTaskForTCB = Task::GetTask( pTCB );
```



Task::IsCurrent

inline

BOOL

Task::IsCurrent()

const;

Returns TRUE if this is the current executing Task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's current task pointer service.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	FALSE if this is NOT the current executing Task object.
TRUE	TRUE if this is the current executing Task object.

Example

```
// you would use this member to determine
// if the given task object is the currently
// executing one, from an interrupt for example

void
MyLowlevelInterrupt::Entry()
{
    // we assume a member in MyLowlevelInterrupt
    // called pMyTask which points to a particular
    // task instance that we are associated with

    if ( pMyTask->IsCurrent() )
    {
        // yes, we interrupted our task
    }
    else
    {
        // no, we interrupted some other task
    }
}
```



Task::PreemptionOff

static

inline

OPTION

Task::PreemptionOff();

Turns preemption off for the currently executing task.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's change preemption task service.
Post-condition	The task's preemption posture is disabled. NOTE: any time-slice associated with the task is disabled. Returns old preemption posture.

Parameters

None

Return Value

Return Value	Overview
validData	The previous preemption posture. Valid options are NU_NO_PREEMPT and NU_PREEMPT.

Example

```

VOID MyTask::Entry()
{
    // do some processing here
    ...

    // here we are going to do something that
    // requires that we are not preemptable by any
    // other tasks.
    Task::PreemptionOff();

    // here even a higher priority task that is
    // ready will not preempt us, we will run
    // exclusively until we suspend or turn
    // preemption back on. NOTE: turning preemption
    // off does not turn interrupts off, they can
    // still occur!

    Task::PreemptionOn(); // turn preemption back on
    ...
}
    
```



Task::PreemptionOn

static

inline

OPTION

Task::PreemptionOn();

Turns preemption on for the currently executing task.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's change preemption task service.
Post-condition	The task's preemption posture is enabled. Returns old preemption posture.

Parameters

None

Return Value

Return Value	Overview
validData	The previous preemption posture. Valid options are NU_NO_PREEMPT and NU_PREEMPT.

Example

See example for PreemptionOff.



Task::Relinquish

static

inline

VOID

Task::Relinquish();

Allows all other ready tasks of the same priority a chance to execute before the currently executing Task object runs again. Relinquishes control to Task objects of identical priority.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's relinquish task service.
Post-condition	The task will allow all other ready tasks of the same priority a chance to execute before this task runs again.

Parameters

None

Return Value

None

Example

```
// in this example, we assume that we have a number of different worker
// tasks that are all at the same priority and there are no tasks of
// lower priority that ever need to run.
void CommWorkerTask::Entry()
{
    while( 1 )
    {
        // check to see if there is something for
        // us to do. If there is then do it.
        if ( CheckForCommand() )
        {
            HandleCommand();
        }

        // allow other tasks at our priority level
        // to run

        Task::Relinquish();
    }
}
```



Task::Reset

STATUS

Task::Reset

```
(
    UNSIGNED suspend = NU_SUSPEND
);
```

Resets this previously terminated or finished `Task` object. The task has a chance to not reset. This member does not resume the task.

Overview

Condition	Description
Pre-condition	The Task object must be previously terminated or finished.
Action	Uses the kernel's reset task service.
Post-condition	If this is a self-resetting task, the function never returns, and therefore will not return anything.

Parameters

Parameter	Overview
suspend	Specifies whether or not to suspend the calling task if the task cannot be reset. If suspend is <code>NU_NO_SUSPEND</code> , the service returns immediately regardless of whether or not the request can be satisfied. Otherwise, suspend contains a timeout value between 1 and 4,294,967,293. The calling task is suspended until the task can be reset, or until the specified number of timer ticks (timeout value) have expired.

Return Value

Return Value	Overview
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.
<code>NU_NOT_TERMINATED</code>	Indicates the specified task is not in a terminated or finished state. Only tasks in a terminated or finished state can be reset.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```
// example 1: a self resetting task. You could use this approach if you
// wanted worker tasks to recycle themselves for a server task that
// spawns them.

void UnderpaidWorkerTask::Entry()
{
    BOOL done = FALSE;

    while( !done )
    {
        // do whatever work we need to do here
    }
    // Reset ourselves so that the task that
    // manages us can go ahead and put us to
    // work again by just calling Start

    Reset();
}

//*****
// example 2: a worker task that just finishes, and a manager task that
// delegates jobs to the workers as they come up. Here, the manager is
// in charge of resetting the workers after they are finished and
// keeping track of what they are doing.

VOID WorkerTask::Entry()
{
    while( !done )
    {
        // do work here
    }

    // we are done, return from the entry routine
}

VOID ManagerTask::Entry()
{

```




```
WorkerTask* pWorker;  
Job* pJobForTheWorker;  
while( 1 )  
{  
    // suspend here waiting for work to come  
    // up for the worker tasks to do  
  
    pJobForTheWorker = WaitForAJob();  
    // you got a job for the worker to do.  
    // get an idle worker task and get him  
    // started on the job  
    pWorker = GetAvailableWorker();  
    pWorker->Reset();  
    pWorker->SetJob( pJobForTheWorker );  
    pWorker->Start();  
}
```



Resume Task::

```
inline  
STATUS  
Task::Resume();
```

Resumes this previously suspended Task object.

Overview

Condition	Description
Pre-condition	Task must be unconditionally suspended, reset, or not yet started.
Action	Uses the kernel's resume task service.
Post-condition	If successful, the task is resumed.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_RESUME	Indicates the specified task is not in an unconditional suspended state.
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
MyWorkerTask* pWorkerTask;  
  
// here the worker task is assumed to be either  
// suspended unconditionally, reset or not yet  
// started.  
  
pWorkerTask->Resume();
```



Task::Sleep

```

static
inline
VOID
Task::Sleep
(
    UNSIGNED ticks
);

```

Suspends current executing task for specified timer ticks.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's sleep task service.
Post-condition	The calling task will sleep for the specified number of timer ticks.

Parameters

Parameter	Overview
ticks	The number of ticks to sleep.

Return Value

None



Example

```
// This is a very useful but sometimes tricky member
// because it is static.

void IncessantBeepTask::Entry()
{
    while( 1 )
    {
        BeepOn(); // turn the sound hardware on
        Task::Sleep( systemClockFrequency / 8 );
        BeepOff(); // turn the hardware off

        // wait for a half second until the next
        // beep

        Task::Sleep( systemClockFrequency / 2 );
    }
}

//*****
// Another example, but this code may not do
// what you might expect!

class MyTask1;
class MyTask2;

void MyTask2::Entry()
{
    // assume a member pointer to a MyTask1
    // object that was setup in the constructor

    pMyTask1->Sleep( 100 );

    // what task slept here? It was actually the
    // MyTask2 instance that slept, not the MyTask1
    // object referred to by pMyTask1. This is because
    // Sleep is a static member and causes whatever
    // thread called it to sleep.
```



Task::Start

inline

STATUS

Task::Start();

Starts this Task object.

Overview

Condition	Description
Pre-condition	Task must be unconditionally suspended, reset, or not yet started.
Action	Uses the kernel's start task service.
Post-condition	The task is started.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_RESUME	Indicates the specified task is not in an unconditional suspended state.
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```

MyWorkerTask* pWorkerTask;

// create a new MyWorkerTask object from the
// global heap.

pWorkerTask = new MyWorkerTask;

// Now start it.  If this is called on a multitasking
// thread, the MyWorkerTask::Entry routine will run
// pWorkerTask is the highest priority ready thread as
// determined by the scheduler.

pWorkerTask->Start();

```



Task::Suspend

inline

STATUS

Task::Suspend();

Unconditionally suspends this Task object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's suspend task service.
Post-condition	If the task was already suspended, the task stays suspended even after its original cause for suspension is lifted. Resume() must be used to resume this task.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
void MyWorkerTask::Entry()
{
    // do whatever we need to do here.

    // we are finished, go ahead and suspend.
    Suspend();
}
```



Task::Task

Task::Task

```
(
const CHAR* name,
    UNSIGNED stacksize,
    OPTION priority,
    UNSIGNED
timeslice,
    BOOL preEmption,
    CHAR* stack = NULL
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create task service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The task's name.
stacksize	Represents the number of bytes for the task stack.
priority	Specifies a priority value between 0 and 255. The lower the numeric value, the higher the priority.
timeslice	Indicates the maximum number of clock ticks that can expire while executing this task. A value of zero for timeslice disables time slicing for this task.
preEmption	TRUE indicates that the task is preemptable, while FALSE indicates that the task is not preemptable. Time slicing is disabled if the task is not preemptable.
stack	If supplied, the task will use the area pointed to as the stack for this task object. Otherwise, the stack is allocated from the free store.

Return Value

None



Example

```
// here we create a Task derived class called MyTask.
class MyTask : public Task
{
    protected:

        virtual void Entry();

    public:

        MyTask();

        virtual
        ~MyTask();
};

MyTask::MyTask()
:
    Task
    (
        "MyTask",    // Task name.
        2048,        // Stacksize.
        10,          // Priority.
        5,           // Timeslice.
        TRUE         // Preemption is on.
    )
{
    // do our specific construction here
}

// later in some other code: The call to new on a MyTask object will
// call the MyTask constructor which in turn will call the Base class
// Task constructor. We must call Start for the task to actually begin.

MyTask* pMyTask = new MyTask;
pMyTask->Start();
```



Task::Terminate

STATUS

Task::Terminate

```
(
    UNSIGNED suspend = NU_SUSPEND
);
```

Terminates this Task object. Derived Task objects that must coordinate termination for elegance should provide a virtual Terminate function.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's terminate task service.
Post-condition	If successful, Task is terminated. A terminated task cannot execute again until it is Reset.

Parameters

Parameter	Overview
suspend	<p>NU_NO_SUSPEND, the service returns immediately regardless of whether or not the request can be satisfied.</p> <p>Otherwise, suspend contains a timeout value between 1 and 4,294,967,293. The calling task is suspended until the task can be terminated or until the specified number of timer ticks (timeout value) have expired. The default value for suspend is NU_SUSPEND.</p>

Return Value

Return Value	Overview
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.



Example

```
// example 1: self terminating tasks. One nice feature of Nucleus
// C++ is that there is optional support for self termination
// of a task. (this is accomplished via another helper task that does
// the actual termination, but you are shielded from that detail...)

VOID MyWorkerTask::Entry()
{
    // slave away doing work here

    Terminate(); // terminate ourselves
}

// example 2: manually Terminating a task in progress say you had a
// worker task that was transmitting a file or something and you wanted
// to cancel the operation by terminating the worker task.

WorkerTask* pWorkerTask = new WorkerTask;
pWorkerTask->Start();
...

// the worker is working away, and then you decide that you want to kill
// the task. Here we attempt to terminate the task and then wait for up
// to 100 ticks for it to terminate. If state of the worker task at
// time of termination is a concern, Tasks have an optional semaphore
// data member called terminationCoordinator that should be
// obtained internally in the worker any time that termination is not
// OK, then released when it is OK.

pWorkerTask->Terminate( 100 );
```



Task::UpdateInfo

```

inline
TaskInfo&
Task::UpdateInfo()
const;

```

Updates internal object information, and returns a `const` reference to the `info` object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information task service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	A reference to the <code>info</code> object.



Example

```
// say we are in a worker management task that
// contains an array of worker tasks. It wants to
// find one that is in either a finished or
// terminated state that it can recycle. This
// information is available via the UpdateInfo call.

WorkerTask*
ManagerTask::FindAvailableWorker()
{
    DATA_ELEMENT status;

    for (int index=0; index < MAX_WORKERS; index++)
    {
        // get updated info on the current worker
        // from the kernel
        const TaskInfo& info =
            workers[index].UpdateInfo();

        // get the task status from the info object
        status = info.GetTaskStatus();

        // we are looking for tasks that are
        // either finished or terminated

        if ((status == NU_FINISHED) ||
            (status == NU_TERMINATED))
        {
            // here is an available worker
            return( &workers[index] );
        }
    }

    // you didn't find an available worker.
    return( NULL );
}
```



class TaskInfo : public ThreadInfo

Derived ThreadInfo class that holds various information about Task objects.

Public Member Functions

Member	Overview
~TaskInfo	Destructor
GetControlBlock	Returns a const reference to the kernel's queue control block for the Task this TaskInfo object is associated with.
GetPreemption	Returns this Task object's preemption option.
GetTaskStatus	Returns the current status of this Task object.
GetTimeslice	Returns this Task object's current time slice parameter.
TaskInfo	Constructor
Update	Updates internal object information, and returns a reference to the info object.



TaskInfo::~TaskInfo

virtual

```
TaskInfo::~TaskInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition:	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



TaskInfo::GetControlBlock

```
inline
NU_TASK&
TaskInfo::GetControlBlock()
const;
```

Returns a const reference to the kernel's queue control block for the Task this TaskInfo object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidReference	A const reference to the kernel's queue control block for the Task this TaskInfo object is associated with.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();

const NU_TASK& tcb = info.GetControlBlock();
```



TaskInfo::GetPreemption

```
inline
OPTION
TaskInfo::GetPreemption()
const;
```

Returns this Task object's preemption option.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The preemption posture. Valid options are NU_NO_PREEMPT and NU_PREEMPT.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();

OPTION preemption = info.GetPreemption();
```



TaskInfo::GetTaskStatus

```
inline
DATA_ELEMENT
TaskInfo::GetTaskStatus()
const;
```

Returns the current status of this Task object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
NU_DRIVER_SUSPEND	Suspended from an I/O Driver request.
NU_EVENT_SUSPEND	Suspended on an event-flag group.
NU_FINISHED	Returned from the entry function.
NU_MAILBOX_SUSPEND	Suspended on a mailbox.
NU_MEMORY_SUSPEND	Suspended on a dynamic-memory pool.
NU_PARTITION_SUSPEND	Suspended on a memory-partition pool.
NU_PIPE_SUSPEND	Suspended on a pipe.
NU_PURE_SUSPEND	Unconditionally suspended.
NU_QUEUE_SUSPEND	Suspended on a queue.
NU_READY	Ready to execute.
NU_SEMAPHORE_SUSPEND	Suspended on a semaphore.
NU_SLEEP_SUSPEND	Sleeping.
NU_TERMINATED	Terminated.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();

DATA_ELEMENT state = info.GetTaskStatus();
```



TaskInfo::GetTimeslice

```
inline  
UNSIGNED  
TaskInfo::GetTimeslice()  
const;
```

Returns this Task object's current time slice parameter.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	This Task object's current time slice parameter.

Example

```
// assume that Task* pTask exists.  
const TaskInfo& info = pTask->UpdateInfo();  
  
UNSIGNED slice = info.GetTimeslice();
```



TaskInfo::TaskInfo

`TaskInfo::TaskInfo();`

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that Task* pTask exists.  
TaskInfo info = pTask->UpdateInfo();
```



TaskInfo::Update

virtual

STATUS

TaskInfo::Update();

Updates internal object information, and returns a reference to the info object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information task service.
Post-condition	If successful, the data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Task* pTask exists.
TaskInfo info = pTask->UpdateInfo();
...
// get the latest.
info.Update();
```



class ThreadInfo : public NucleusPlusInfo

Derived NucleusPlusInfo class that holds various information about Nucleus C++ objects that have associated execution threads. Since this class contains a pure virtual function, objects of this class cannot be declared.

Public Member Functions

Member	Overview
~ThreadInfo	Destructor
GetMinimumStackSize	Returns the minimum amount of bytes left in this Task object.
GetPriority	Returns the current priority of this Task object.
GetStackAddress	Returns a pointer to this Task object's beginning of stack.
GetStackSize	Returns this Task object's stack size.
GetTimesScheduled	Returns the number of times task has been scheduled.
ThreadInfo	Constructor



ThreadInfo::~ThreadInfo

virtual

```
ThreadInfo::~ThreadInfo();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



ThreadInfo::GetMinimumStackSize

```
inline
UNSIGNED
ThreadInfo::GetMinimumStackSize()
const;
```

Returns the minimum amount of bytes left in this `Task` object's stack.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The minimum amount of bytes left in this <code>Task</code> object's stack.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();
```



ThreadInfo::GetPriority

inline

OPTION

ThreadInfo::GetPriority()

const;

Returns the current priority if this Task object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current priority if this Task object.

Example

```
// assume that Task* pTask exists.  
const TaskInfo& info = pTask->UpdateInfo();
```



ThreadInfo::GetStackAddress

```
inline
VOID*
ThreadInfo::GetStackAddress()
const;
```

Returns a pointer to this Task object's beginning of stack.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A pointer to this Task object's beginning of stack.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();
VOID* pStack = info.GetStackAddress();
```



ThreadInfo::GetStackSize

```
inline  
UNSIGNED  
ThreadInfo::GetStackSize()  
const;
```

Returns this Task object's stack size.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	This Task object's stack size.

Example

```
// assume that Task* pTask exists.  
const TaskInfo& info = pTask->UpdateInfo();  
  
UNSIGNED stacksize = info.GetStackSize();
```



ThreadInfo::GetTimesScheduled

```
inline
UNSIGNED
ThreadInfo::GetTimesScheduled()
const;
```

Returns the number of times task has been scheduled.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of times task has been scheduled.

Example

```
// assume that Task* pTask exists.
const TaskInfo& info = pTask->UpdateInfo();
```



ThreadInfo::ThreadInfo

ThreadInfo::ThreadInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

ThreadInfo is a base class shared by TaskInfo and HighLevelInterruptInfo and is not directly used.



class Timer : public NucleusPlus

Nucleus C++ provides the application with programmable timer objects. These timers execute a specific derived object member routine when they expire. The object's expiration routine executes as a high level interrupt service routine. Therefore, self suspension requests are not allowed. Additionally, processing should be kept to a minimum. A Timer object provides a mechanism to execute a timer routine when they expire. The routine executes as a high-level interrupt service routine. Therefore, self-suspension requests are not allowed. Processing should be kept to a minimum in derived Timer object expiration routines.

Public Member Functions

Member	Overview
~Timer	Destructor
Disable	Disables this Timer object.
Enable	Enables this Timer object.
ExpirationRoutine	Virtual expiration routine to call upon the kernel's timer expiration service. The base class member is a pure virtual. Derived timer objects will provide their own timer expiration routine.
Reset	Resets this Timer object to new operating parameters.
Timer	Constructor
UpdateInfo	Updates internal object information, and returns a reference to the info object.



Timer::~Timer

virtual

```
Timer::~Timer();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's delete timer service.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



Timer::Disable

inline

STATUS

Timer::Disable();

Disables this Timer object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's disable timer service.
Post-condition	The timer is disabled.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_ENABLE	Indicates the <code>enable</code> parameter is invalid.
NU_INVALID_TIMER	Indicates the <code>timer</code> pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Timer* pMyTimer exists.

pMyTimer->Disable();
```



Timer::Enable

inline

STATUS

```
Timer::Enable();
```

Enables this Timer object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's enable timer service.
Post-condition	The timer is enabled.

Parameters

None

Return Value

Return Value	Overview
NU_INVALID_ENABLE	Indicates the <code>enable</code> parameter is invalid.
NU_INVALID_TIMER	Indicates the timer pointer is invalid.
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// assume that Timer* pMyTimer exists.  
  
pMyTimer->Enable();
```



Timer::ExpirationRoutine

virtual

VOID

Timer::ExpirationRoutine()=0;

Virtual expiration routine to call upon the kernel's timer expiration service. The base class member is a pure virtual. Derived timer objects will provide their own timer expiration routine.

Overview

Condition	Description
Pre-condition	Derived class specific behavior.
Action	Derived class specific behavior.
Post-condition	Derived class specific behavior.

Parameters

None

Return Value

None

Example

```
// This is the virtual function that you overload
// to do work in your derived timer class.

void MyTimer::ExpirationRoutine()
{
    // do your stuff here
}
```



Timer::Reset

STATUS

Timer::Reset

```
(
    UNSIGNED initialTime,
    UNSIGNED rescheduleTime,
    BOOL enabled
);
```

Resets this `Timer` object to new operating parameters. By default, the current operating parameters are used.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's reset timer service.
Post-condition	If successful, the timer is reset to the supplied input parameters.

Parameters

Parameter	Overview
<code>initialTime</code>	The initial number of timer ticks for timer expiration.
<code>rescheduleTime</code>	The number of timer ticks for expiration after the initial expiration. If this parameter is zero, the timer only expires once.
<code>enabled</code>	TRUE if timer is initially enabled.

Return Value

Return Value	Overview
<code>NU_INVALID_ENABLE</code>	Indicates the enable parameter is invalid.
<code>NU_INVALID_FUNCTION</code>	Indicates the expiration function pointer is NULL.
<code>NU_INVALID_TIMER</code>	Indicates the timer pointer is invalid.
<code>NU_NOT_DISABLED</code>	Indicates the timer is currently enabled. It must be disabled before it can be reset.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```
// assume that MyTimer* pMyTimer exists.  
  
// reset the timer to go off initially after 1  
// second, not go off again, and be initially  
// enabled.  
pMyTimer->Reset( systemClockFrequency, 0, TRUE );
```



Timer::Timer

Timer::Timer

```
(
const CHAR* name,
    UNSIGNED initialTime,
    UNSIGNED rescheduleTime,
    BOOL
enabled
);
```

Constructor

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's create timer service.
Post-condition	The object exists.

Parameters

Parameter	Overview
name	The name of the Timer.
initialTime	The initial number of timer ticks for timer expiration.
rescheduleTime	The number of timer ticks for expiration after the initial expiration.
enabled	TRUE if timer is initially enabled.

Return Value

None

Example

```
// Create a timer that has an initial period of
// 1000 ticks, and then 100 after that. Make
// it initially enabled.
Timer* pTimer1 = new Timer("t1",1000,100,TRUE);
// Create a one shot timer that has an initial
// period of 100 ticks, and then does not fire
// again. Make it initially disabled.
Timer* pTimer2 = new Timer("t1",100,0,FALSE);
```



Timer::UpdateInfo

TimerInfo&

Timer::UpdateInfo() const;

;

Updates internal object information, and returns a reference to the info object.

Overview

Condition	Description
Pre-condition	None
Action	Uses the kernel's information timer service.
Post-condition	The <code>info</code> member has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	A reference to the <code>info</code> object.

Example

```
// assume that MyTimer* myTimer exists.

const TimerInfo& info = myTimer->UpdateInfo();
```



class TimerInfo : public NucleusPlusInfo

Derived NucleusPlusInfo class that holds various information about Timer objects.

Public Member Functions

Member	Overview
~TimerInfo	Destructor
GetControlBlock	Returns a const reference to the kernel's queue control block for the object.
Tmer	This TimerInfo object is associated with.
GetExpirations	Returns the number of times this Timer has expired.
GetId	Returns a pointer to the derived Timer object.
GetInitialTime	Returns this Timer object's initial time value.
GetRescheduleTime	Returns this Timer object's current reschedule time.
IsEnabled	Returns TRUE if this Timer object is currently enabled.
TimerInfo	Constructor
Update	Updates the data members.



TimerInfo::~TimerInfo

virtual

TimerInfo::~TimerInfo();

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



TimerInfo::GetControlBlock

```
inline  
NU_TIMER&  
TimerInfo::GetControlBlock()  
const;
```

Returns a `const` reference to the kernel's queue control block for the `Timer` this `TimerInfo` object is associated with.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>aValidReference</code>	A <code>const</code> reference to the kernel's queue control block for the <code>Timer</code> this <code>TimerInfo</code> object is associated with.

Example

```
// assume that derived class MyTimer* pTimer exists.  
  
// get the latest info on the timer from the kernel.  
const TimerInfo& info = pTimer->UpdateInfo();  
  
// gets the reference to the actual control block  
const NU_TIMER& tcb = info.GetControlBlock();
```



TimerInfo::GetExpirations

```
inline
UNSIGNED
TimerInfo::GetExpirations()
const;
```

Returns the number of times this Timer has expired.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The number of times this Timer has expired.

Example

```
// assume that derived class MyTimer* pTimer exists.
UNSIGNED expirations;
// update the info object, get a const reference
// to it, and get the expirations
expirations = pTimer->UpdateInfo().GetExpirations();
```



TimerInfo::GetId

```
inline
UNSIGNED
TimerInfo::GetId()
const;
```

Returns the user supplied id. In this implementation, this field is used to house a pointer to the derived Timer object.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The user supplied id.

Example

```
// assume that MyTimer* myTimer exists.
const TimerInfo& info = myTimer->UpdateInfo();
Timer* pTimer = (Timer*)info.GetId();
// check to make sure the pointers are
// the same
NU_ASSERT( pTimer == myTimer );
```



TimerInfo::GetInitialTime

```
inline
UNSIGNED
TimerInfo::GetInitialTime()
const;
```

Returns this Timer object's initial time value.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	This Timer object's initial time value.

Example

```
// assume that derived Timer MyTimer* myTimer exists.
const TimerInfo& info = myTimer.UpdateInfo();
UNSIGNED time = info.GetInitialTime();
```



TimerInfo::GetRescheduleTime

```
inline  
UNSIGNED  
TimerInfo::GetRescheduleTime()  
const;
```

Returns this Timer object's current reschedule time.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	This Timer object's current reschedule time.

Example

```
// assume that derived Timer MyTimer* myTimer exists.  
const TimerInfo& info = myTimer.UpdateInfo();  
UNSIGNED time = info.GetRescheduleTime();
```



TimerInfo::IsEnabled

```
inline
BOOL
TimerInfo::IsEnabled()
const;
```

Returns TRUE if this `Timer` object is currently enabled.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	FALSE if this <code>Timer</code> object is NOT currently enabled.
TRUE	TRUE if this <code>Timer</code> object is currently enabled.

Example

```
// assume that derived class MyTimer* pTimer exists.
// get the latest info on the timer from the kernel
// and assign it to the info object we just created.
TimerInfo info = pTimer->UpdateInfo();
BOOL bIsEnabled = info.IsEnabled();
// or you could just do
bIsEnabled = pTimer->UpdateInfo().IsEnabled();
```



TimerInfo::TimerInfo

TimerInfo::TimerInfo();

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// assume that derived class MyTimer* pTimer exists.  
// get the latest info on the timer from the kernel  
// and assign it to the info object we just created.  
  
TimerInfo info = pTimer->UpdateInfo();
```



TimerInfo::Update

virtual

STATUS

TimerInfo::Update();

This function is used to update the data members.

Overview

Condition	Description
Pre-condition	None
Action	Updates the information object for the <code>Timer</code> object.
Post-condition	If successful, the data in the object has been updated and contains timely data.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_TIMER</code>	Indicates the timer pointer is invalid.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// assume that derived class MyTimer* pTimer exists.
// get the latest info on the timer from the kernel
// and assign it to the info object we just created.
TimerInfo info = pTimer->UpdateInfo();
...
// update the info to the latest.
info.Update();
```



